Parallelization of the 2D Roe Scheme on the Intel Paragon

THESIS

John R. Graham III
Captain, USAF

AFIT/GCS/ENG/94D-04

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base

AFIT/GCS/ENG/94D-04

Parallelization of the 2D Roe Scheme on the Intel Paragon

THESIS

John R. Graham III
Captain, USAF

AFIT/GCS/ENG/94D-04

AFIT/GCS/ENG/94D-04

Parallelization of the 2D Roe Scheme on the Intel Paragon

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

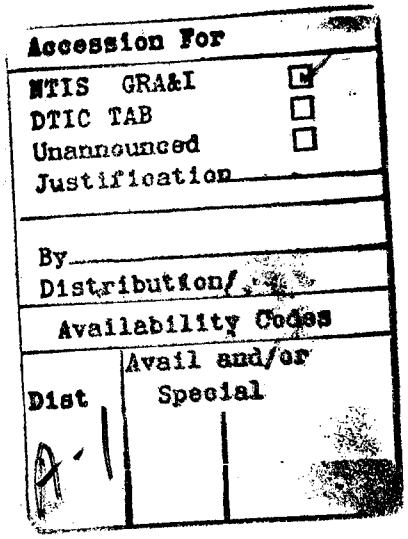
Requirements for the Degree of

Master of Science in Computer Engineering

John R. Graham III, B.S.C.S

Captain, USAF

December 1994



Working in the area of Computational Fluid Dynamics proved to be quite a challenge for me. First, I would like to thank my thesis advisor, LtCol Hobart, for his support, guidance, an unwavering enthusiasm. His timely pats on the back and words of wisdom made this challenge more bearable. Thanks also to my thesis committee, Dr Lamont, Dr Beran, and Maj Doty for their time, support, and work that they put into making my thesis a success. I would also like to thank Doug Blake for his patience and willingness to take the time to help me understand the fundamentals of aerodynamics and CFD.

Most of all, I would like to thank my wonderful wife for her resilience, understanding, and untiring support during our first year of marriage which unfortunately coincided with the thesis "crunch time."

John R. Graham III

Table of Contents

List of Figures

AFIT/GCS/ENG/94D-04

Abstract

This study presented a methodology for determining the general performance characteristics of a computational fluid dynamics (CFD) algorithm on the Intel Paragon. By performing a rigorous time complexity analysis of a parallel CFD algorithm, the general performance could be characterized before the code was actually parallelized. This was shown by implementing a serial version of the 2-D Roe Scheme on the Paragon. This explicit code was parallelized by the addition of generic yet efficient routines that decomposed the domain, automatically adjusted partition indices, and performed 2-D and 3-D buffer exchanges. Additionally, efficient global routines available for the Paragon were used in order to reduce the overall complexity of the parallel implementation. Comparison of the predicted performance and the measured performance showed that for the Roe Scheme, the general performance characteristics on the Intel Paragon could be accurately predicted. While a complexity analysis of the Beam-Warming algorithm was performed, a working parallel implementation on the Paragon was not completed.

Parallelization of the 2-D Roe Scheme on the Intel Paragon

I. Introduction

Theoretical performance capabilities of sequential computers have provided a generally decreasing rate of return with the introduction of each new family of vector processor supercomputers. In contrast, massively parallel architectures have shown a significant increase in performance capabilities over the past few years. Figure 1 shows the theoretical performance of many of today's popular supercomputers.

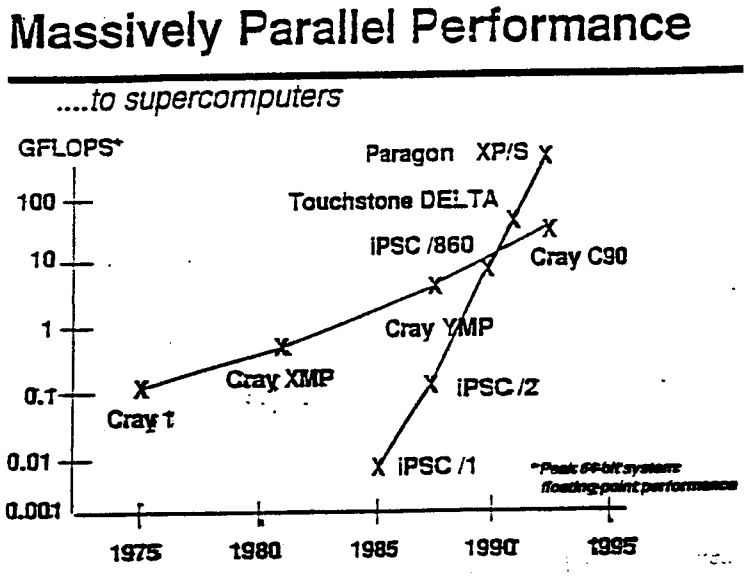


Figure 1. Computing Capacity of Today's Supercomputers

The deceasing cost of processor technology, the nonlinear relationship between a problem's size and the computing power needed to solve it, and the physical limit on serial processor performance (the speed-of-light argument) (26) are all driving the supercomputing trend towards scalable architectures. Scalable architectures offer the best hope of

achieving the 1 TFLOPS performance of Grand Challenge type problems. For example,

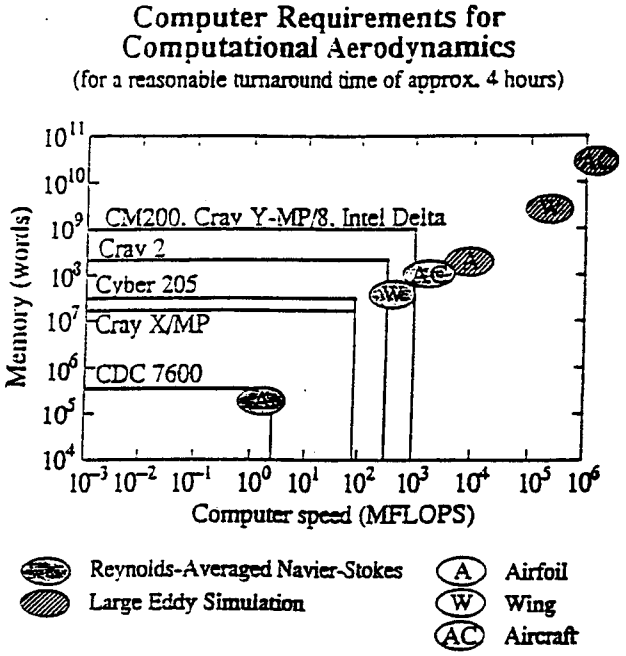


Figure 2. Computing Capacity Needed for Typical Problems

Figure 2(22) shows the estimated performance needed to support Grand Challenge problems associated with computational aeromechanics. The spectrum of Grand Challenge problems is shown in Figure 3 (22).

One challenging application area with a huge potential return is computational fluid dynamics (CFD). CFD is the translation of fluid dynamics problems to a computational or numeric representation. The associated "model" represents the behavior of the fluid within the given problem space or system. Because these problems are so large and complex, the implementation of these computational models has historically consumed the largest computational resources available. Today, CFD solutions are able to model complex fluid flow around an object with a simple geometry, or simple fluid flow around an object with a complex geometry, but not both (12).

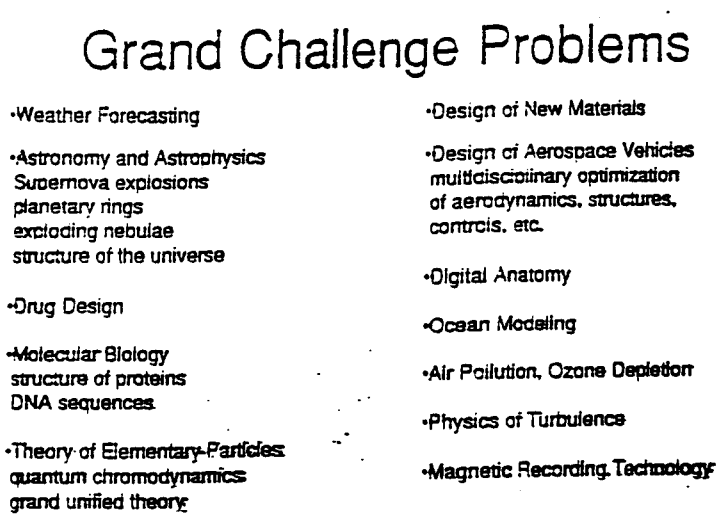


Figure 3. Grand Challenge Problems

Recent advances in parallel processing architectures, algorithms, software, and compilers have pushed the performance of some parallel CFD solutions within reach of the capability of current leading-edge vector based supercomputers.

1.1 Problem Statement

Numerous parallel techniques in the areas of load balancing (e.g., recursive spectral bisection, pairwise cyclic exchange, etc.), domain decomposition, and parallel flow solvers have been developed. However, little work has been done to develop a methodology for implementation of parallel CFD algorithms. Additionally, few researchers have used quantitative measurements to compare the predicted general performance of a CFD algorithm with the measured performance.

1.2 Approach and Scope

The solution to the above problem was to develop a methodology for characterizing the general performance of a parallel CFD implementation. Routines were generated to parallelize a serial CFD code and an estimate on the general performance was derived based on the complexity analysis of the algorithms used. Processing times were measured and compared to the estimates for validation.

The methodology, parallel routines, and measurements were first applied to an explicit algorithm for inviscid, supersonic flow over a cylinder. A secondary objective was to apply the same methodology and measurements to an implicit solver.

The target parallel computer architecture was the Intel Paragon. The explicit and implicit solvers studied were the Roe and Beam-Warming algorithms respectively. Grids were generated using structured grid generation techniques.

Contemporary research in parallel CFD algorithms is discussed in Chapter 2. Chapter 3 presents the governing equations, boundary conditions, and serial algorithm analysis and Chapter 4 details the parallel algorithm and parallelization issues of the Roe Scheme on the Intel Paragon. Chapter 5 presents measurements of the performance of the Roe Scheme and compares the estimated performance to the measured performance. Finally, conclusions and recommendations for future research is provided in Chapter 6.

II. Background

CFD problems are among the most demanding scientific computing problems in terms of the required computational resources. Increased problem size and complexity has taxed traditional vector supercomputer capacities and capabilities. With the introduction of massively parallel processing (MPP) systems, each computer system can bring more computational resources to bear on a specific application and thus reduce the overall solution time. The purpose of this chapter is to examine literature dealing with the integral parts of a CFD solution with emphasis on their use in MPP message passing architectures such as the Intel Paragon. Literature researched is presented as it relates to the particular parts of a parallel CFD solution. The topic areas include gridding methods, domain decomposition, flow solvers, and load balancing.

2.1 Applicability of Parallel Computers for CFD

Historically, complex CFD problems have been solved on vector supercomputers. However, these computer architectures do not have the computational power to solve CFD problems requiring 3-D modeling of unsteady and viscous flows (17). Several authors have shown that massively parallel computers can equal or better the performance of a vector machine for CFD problems. This section discusses the integral parts of a CFD solution and presents how parallel computers can be applied to effectively solve a particular piece of the overall problem. Additionally, results from research on various parallel machines is presented.

2.2 Grid Generation Methods

There are two primary methods of grid generation: structured and unstructured. Because complex boundaries do not occur in one-dimensional problems and there are numerous functions which can be used to generate a two-dimensional structured grid, most grid generation work has focused on creating structured grids in two dimensions. As a result, methods for generating two-dimensional structured grids are well known and classified as follows:

1. Complex variable methods,
2. Algebraic methods, and
3. Differential equation techniques.

The physical grid is transformed to a uniformly spaced computational grid by implementation of one of these methods. Anderson et al. (2) give an excellent and comprehensive description of algebraic and differential equation techniques. Discussion of methods for generating three-dimensional grids is beyond the scope of this thesis.

Limited research in the area of parallel grid generation has been performed. Because structured grid generation is a relatively mature field and these grids are efficiently generated on sequential processors. However, little information exists on parallel structured grid generation. Due to the inherent parallelism in unstructured grid generation, some research into unstructured grid generators has been performed.

In a recent paper, Lönher, Cambros, and Merriam (24) showed the ability to parallelize the grid generation process by implementing an unstructured grid generator using

the advancing front method. The researchers state that introduction of a point requires checking locally for compatibility, thus enabling several points to be introduced in parallel.

Research performed by Davis (13) modified a sequential 2-D unstructured grid generator developed by Smith (30). Davis found that a parallel grid generator using Delauney triangulation provided little increase in performance. In fact, due to synchronization costs and additional coding to provide deadlock avoidance, using Delauney triangulation hindered performance for multiple processors.

2.3 Domain Decomposition

The potential of parallelization of large scale problems has caused great interest to be placed on domain decomposition methods (DDMs). DDMs allow an original domain to be subdivided, or decomposed, into smaller subdomains. This allows simultaneous solution of the subdomains that make up the original problem.

Chan applied an overlapped domain decomposition technique to two types of problems. The first problem was a class of convection-diffusion problems in two dimensions and the second, a two dimensional driven cavity problem. Chan found that the rate of convergence decreased exponentially as the amount of overlap was reduced (8).

A quantitative analysis of parallel efficiency of domain decomposition methods was explored by Hoffman and Zou. The authors analyzed a parallel Schwarz-type, an additive Schwarz-type, and an iterative substructuring DDM by Bramble et al. (7), Zhang and Huang (36), and Dryja and Widlund (14). Zhang and Huang's parallel Schwarz-type DDM was projected to have superlinear speedup. The additive Schwarz-type and iterative

substructuring DDMs were projected to be inefficient for more than 64 processors and 256 processors respectively. While the research shows the promise of parallel implementation for these types of DDMS, no comparisons to performance on an actual parallel processing architecture was made.

In another paper, Farhat and Roux propose a DDM that requires interprocessor communication during solution of the interface problem and uses nearest neighbor communications in assembling the subdomain results (15). As a result, the amount of message passing is reduced and the complexity of the communication is reduced creating a more effective mapping of subdomains to processors. While the authors' DDM converged two times slower than a subdomain-preconditioned Schur method, the DDM provided a speedup of 28.8 for a 32-processor partition and was 16% faster than the subdomain preconditioned Schur method.

Gropp and Keyes developed complexity estimates for the performance of partitioned matrix methods applied to preconditioned conjugate gradient techniques for shared and distributed memory parallel computers (16). The authors investigated both strip and block decomposition and considered only DDM methods that employ nearest neighbor communication. Block decomposition provided the best performance in tests performed on an iPSC Hypercube.

An intriguing approach to domain decomposition was research performed by Angelaccio and Colajanni on subcube matrix decomposition(3). An effort was made to design decomposition-independent algorithms that would handle LU factorization for dense matrices and without pivoting. The authors propose partitioning a hypercube or mesh

architecture into subcubes (or meshes) in which the matrix entries are mapped onto the subcubes instead of particular nodes. The experimental results on a iPSC/2 hypercube show that for the three different proposed algorithms, the speedup scaled linearly with the number of processors.

Barth's paper (5) presents DDM based on coordinate bisection, Cuthill-McKee, and spectral partitioning algorithms. His findings agree with Kumar (23) that coordinate bisection methods, because no connectivity information is used, produce partitions that give less than optimal communication patterns. While spectral partitioning algorithms were found to yield connected partitions that were well balanced, they tended to be computationally expensive.

A paper by Leete, Peyton, and Sincovec presented an improved RSB technique.

- Implementing a parallel version,
- Using a proportional mapping to handle an arbitrary number of processors, and
- Implementing a more flexible load balancing technique.

The improved technique was used on two test cases: an aircraft/wing configuration with 2,851 vertices and 15,093 edges and a four-element airfoil with 30,269 vertices and 44,929 edges. Performance of the code on a 128-processor Intel iPSC/860 showed an almost even distribution of work as the number of processors was increased. Additionally, the overall execution time was reduced as the number of processors increased from 2 to 32. Thus, this improved method provides the functionality and capability provided by the serial implementation while operating in a parallel environment.

Simon applied coordinate bisection, graph bisection, and spectral bisection algorithms to several two-dimensional and three-dimensional problems (29). The spectral bisection technique produced uniform subdomains with short boundaries and thus yielded better partitions. The coordinate bisection technique left disconnected partitions which lengthened boundary segments and required additional communications. The graph bisection technique produced partitions with long boundaries which resulted in increased communication costs.

In a separate paper, Venkatakrishnan also implemented the three bisection algorithms as part of a parallel Euler solver. Although coordinate bisection and graph bisection took less computation time, they required more overall communication than the spectral bisection method. For a problem containing 15,606 vertices, the spectral method was 25% to 150% faster than the other two methods. Tests were performed using a 64-processor Intel iPSC/860.

A paper by Henderson and Leland describes an algorithm based on spectral bisection that produces an effective load balancing technique for scientific computations. The technique, known as recursive spectral quadsection (RSQ) and recursive spectral octsection (RSO) recursively bisects a graph by considering an eigenvector of an associated matrix in order to understand the global properties of the graph (19). This method was found to be superior to inertial and recursive spectral bisection (RSB) load balancing techniques because of its ability to reduce communications due to ineffective workload distribution among multiple processors.

2.3.1 Flow Solvers. Flow solvers provide the computational algorithm for solving a given set of equations and boundary conditions. While much CFD research has been done for fluid flow on sequential computers, the potential decrease in execution time has caused great interest in parallelization of flow solvers. Typically, Euler and Navier-Stokes equations are used in flow solvers.

Venkatakrishnan, Simon, and Barth proved that an explicit unstructured flow solver using MacCormack's algorithm could be implemented on a MIMD machine with performance obtained by traditional vector supercomputers (32). They found that 128 processors on an iPSC/860 could produce the same performance of a single processor Cray YMP/1 for a small problem size(6,019 vertices) and over twice the performance of the Cray for a large problem size (over 15,600 vertices).

In a recent paper, Das, Mavriplis, Saltz, Gupta, and Ponnusamy design and implement a parallel explicit unstructured Euler solver and compare the performance between an iPSC/860 hypercube, an Intel Touchstone Delta, and a Cray YMP (10). The authors used software tools to automatically optimize the solvers for a parallel architecture. The authors show the parallelization potential of CFD codes by showing that using 128 and 256 node test cases, the Delta outperformed the Cray YMP by a factor of between four and seven. A 128 node iPSC/860 was four times faster than the Cray.

Chyczewski, Marconi, Pelz, and Churchitser implemented an Euler and a Navier-Stokes solver for studying the performance of the Thomas Algorithm on a nCUBE/2 and CM-5 multiprocessor(11). The authors show that the implementation of an approximate factorization technique can improve the performance of an implicit solution by reducing

the communications required to solve tridiagonal matrices. The algorithm executing on a 128 nCUBE/2 system was found to operate as fast as on a Cray. However, the performance of the algorithm on a NxM matrix deteriorated once the number of processors (p) became larger than N or M.

Das, Mavriplis, Saltz, and Vermeland compared the implementation of a parallel unstructured solver, EUL3D, on both shared and distributed memory multiprocessor computers(25). The authors found that implementation on distributed memory machines, such as the Intel Delta, was significantly more difficult than on the Cray Y-MP C90 due to a lack of sophisticated software tools. This was attributed to the relative immaturity of parallel processing support tools. The Intel Delta machine with 512 processors was found to deliver the same performance as a 5 processor Cray Y-MP C90.

In a paper titled "Implicit CFD Applications on Message Passing Multiprocessor Systems", Naik, Naik, and Nicoules profile ARC-3D, which is an implicit flow solver that uses the Beam-Warming approximate factorization algorithm and and is well known among the CFD community. The authors state that implicit schemes have proven to be superior to explicit schemes when performed on uni-processor and vector architectures. However, implementation on message passing architectures requires that data dependencies be taken into account in order to determine the most efficient solution for a given problem. Results from test run on a Victor multiprocessor system showed a speed up over 7, 9, and 12 on a 8, 12, and 16-processor system respectively. Thus, impressive performance of implicit CFD codes can be obtained by implementation on MIMD architectures (27).

On vector supercomputers implicit schemes typically converge to a solution in less time that an explicit scheme. However, on parallel architectures the communications required to solve implicit schemes results in faster convergence for explicit schemes. A paper by Venkatakrishnan presented a comparison the rate of convergence for a parallel implementation of an implicit and explicit scheme on a hypercube architecture (Intel iPSC/860). Subcritical flow past a four-element airfoil with $M_\infty = 0.2$ and an angle of attack of 5° was used as a test case for both methods. He found that implicit schemes can be designed to converge to a steady state faster than explicit schemes by using various methods of preconditioning (33).

Blake's investigation of a parallel implementation of a 1-D explicit Roe Scheme to model the flow field in a shock tube provides an excellent analysis on the theoretical performance as well as the actual performance on an iPSC/2 and a Paragon X/PS. Blake found that the iPSC/2 achieved a speedup of 7.75 with 8 processors and a speedup of 7.5 with 8 processors on a 400×400 grid. Additionally, while the isoefficiency functions of both architectures were nearly identical, the Paragon did not exhibit the speedup characteristics of the hypercube.

Scherr implemented a parallel explicit MacCormack algorithm for solving the Navier-Stokes equations on an Intel Touchstone Delta (28). Test were performed on a 75° swept delta wing for $M_\infty = 1.95$, an angle of attack of 30°, and a Reynolds number of 4.48×10^6. He found that on an $80 \times 80 \times 80$ grid, the parallel implementation with 32 processors outperformed the same algorithm executed on a single processor Cray 2. However, efficiency dropped rapidly as the number of processors increased above 64.

Anderson et al.(2) and Hirsch (20)(21) also give excellent and comprehensive description of computational methods for solving fluid mechanics and heat transfer problems.

2.3.2 Load Balancing. For static grids, the initial domain decomposition methods described above prove adequate for providing a balanced workload for each processor. For dynamic grids, a load balancing algorithm ensures that uneven workload distribution does not inhibit the maximum parallel performance of a given algorithm. Thus, for the workload to be evenly spread across a large number of processors, a mechanism to ensure proper balance of work must exist.

Willebeek-LeMair and Reeves present several strategies for dynamically balancing workloads. They are:

1. Sender initiated diffusion,
2. Receiver initiated diffusion,
3. Hierarchical balancing method,
4. Gradient model, and
5. Dimension exchange method.

The sender initiated diffusion strategy uses nearest-neighbor load information to reapportion workload from heavily loaded areas to lightly loaded areas. In the technique, the overloaded processor (sender) generates the request to lighten its workload. Receiver initiated diffusion is identical to sender-initiated diffusion except the lightly loaded processor generates the workload request. The hierarchical balancing method organizes the system into a hierarchy and balances the workload by propagating subtree imbalances to

the root level. The gradient model uses a gradient map of the proximities of underloaded processors to guide the migration of tasks from heavily loaded processors to lightly loaded processors. Finally, the dimension exchange method balances the workload by iteratively folding an n processor system into $\log n$ dimensions and balancing the workload in each dimension. The performance of each of the above algorithms was measured on a 32-node iPSC/2 a test case. The receiver initiated diffusion, hierarchical balancing method, and dimension exchange method produced the highest speedup while the gradient model and sender initiated diffusion produced the lowest (35).

Walshaw and Berzins present another modification of RSB called dynamic recursive spectral bisection (DRSB) that is more suitable for problems that require frequent remeshing. RSB-based algorithms are typically expensive computationally. As a result, changes to the mesh or grid result in an expensive recalculation of the workload for each processor even if redistribution is not required. DRSB effectively handles the incremental partitioning problem due to a partition being refined and/or coarsened by interpolating the existing partition onto a new partition and using it as a start in a repartitioning algorithm. Because the mesh is unlikely to change a great deal when repartitioned, this interpolated partition typically requires very few changes to achieve the optimal load balance. This quick interpolation greatly reduces the computation time associated with RSB. Tests showed DRSB to be take less time to load balance and evaluate the residuals than RSB (34).

Research by Hammond on load balancing produced an algorithm in which each processor may exchange the tasks mapped to it with nearest-neighbor processors. This cyclic pairwise exchange (CPE) algorithm outperformed methods based on simulated annealing for irregular problems like 2-D flow around multi-component airfoils and 3-D around air-

craft (18). Hammond also found that using CPE to map tasks to processors made the performance of irregular test cases on a 8,192 processor CM-2 similar to the performance achieved on a single processor Cray Y-MP and a 64-processor Intel iPSC/860. While Hammond's experiments were performed on a SIMD architecture (CM-2), the implementation of the algorithm on a mesh architecture such as the Intel Paragon should produce similar communication/computation savings.

2.4 Summary

The research of the authors presented in this chapter provides the background and justification for pursuing parallelization of CFD. The finding by numerous researchers in the different areas related to CFD demonstrate that the goal of reducing the time to provide a complex solution on massively parallel processing machines is possible. The research previously performed provides the background for choosing the most effective methods for parallelization of explicit and implicit CFD codes. Because of their familiarity and wide distribution to the CFD community, the Beam-Warming and Roe schemes were used for providing the implicit and explicit solution respectively. Standard coordinate transformation from the Cartesian coordinate system to the computational domain and block decomposition were used for structured grid solutions.

III. Algorithm Definition

This chapter presents the background information for the study of a flow field given a pgeometry. The governing equations and boundary conditions for the problems are presented as well as an explanation of the discretization method used. Finally, the implicit and explicit algorithms to be parallelized are explained and a serial time complexity analysis is presented.

3.1 Governing Equations

The governing equations for two-dimensional inviscid flow for a two-dimensional Cartesian coordinate system can be written as

$$\frac{\partial U}{\partial t} + \frac{\partial E}{dx} + \frac{\partial F}{dy} = 0 \tag{1}$$

where

$$\underline{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E_t \end{bmatrix} \tag{2}$$

$$\underline{E}(\underline{U}) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E_t + p)u \end{bmatrix} \tag{3}$$

$$\underline{F}(\underline{U}) = \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ (E_t + p)v \end{bmatrix} \tag{4}$$

$$E_t = \rho\left(e + \frac{u^2 + v^2}{2}\right) \tag{5}$$

where U,E, and F are inviscid flux vectors. The flow variables density, pressure, total energy, and velocity are denoted by ρ, P, E_t, U and V respectively.

The conservative form of these equations is used in order to simplify the procedure for implementation of input and boundary conditions. Equation 1 becomes

$$\frac{\partial U^*}{\partial t^*} + \frac{\partial E^*}{dx^*} + \frac{\partial F^*}{dy^*} = 0 \tag{6}$$

where

$$\underline{U^*} = \begin{bmatrix} \rho^* \\ \rho^* u^* \\ \rho^* v^* \\ E^*_{t^*} \end{bmatrix} \tag{7}$$

$$\underline{E^*}(\underline{U^*}) = \begin{bmatrix} \rho^* u^* \\ \rho^* u^{*2} + p^* \\ \rho^* u^* v^* \\ (E^*_{t^*} + p^*)u^* \end{bmatrix} \tag{8}$$

$$\underline{F^*}(\underline{U^*}) = \begin{bmatrix} \rho^* v^* \\ \rho^* u^* v^* \\ \rho^* v^{*2} + p^* \\ (E^*_{t^*} + p^*) v^* \end{bmatrix} \tag{9}$$

$$E^*_{t^*} = \rho^* \left(e^* + \frac{u^{*2} + v^{*2}}{2} \right) \tag{10}$$

Because the system of equations contains five equations and six unknowns, it is necessary to develop an additional equation to solve the linear system of equations. Thus,

$$p^* = (\gamma - 1)\rho^* e^* \tag{11}$$

where γ is the ratio of specific heats of the working fluid.

3.2 The Finite Difference Method

Because a digital computer cannot directly solve a continuous partial differential equation (PDE), the PDE must be approximated. One of the most utilized discretization methods used for solving CFD problems is the finite-difference method.

The finite-difference method can use a Taylor's series expansion to solve the PDEs. Given a function $f(x)$, $f(x + \Delta x)$ can thus be expanded in a Taylor series about x as

$$f(x + \Delta x) = f(x) + (\Delta x)\frac{\partial f}{\partial x} + \frac{(\Delta x)^2}{2!}\frac{\partial^2 f}{\partial x^2} + \frac{(\Delta x)^3}{3!}\frac{\partial^3 f}{\partial x^3} + \dots \tag{12}$$

Solving for $\frac{\partial f}{\partial x}$ yields

$$\frac{\partial f}{\partial x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} - \frac{\Delta x}{2!}\frac{\partial^2 f}{\partial x^2} - \frac{(\Delta x)^2}{3!}\frac{\partial^3 f}{\partial x^3} + \ldots \tag{13}$$

which can be written as

$$\frac{\partial f}{\partial x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} - O(\Delta x) \tag{14}$$

where $O(\Delta x)$ represents terms with factors of Δx and higher. Using $f(x)$ and $f(x+\Delta x)$ is known as a forward difference approximation.

A backward difference approximation results by applying the same approximation in the reverse direction. Thus using $f(x)$ and $f(x-\Delta x)$ yields

$$\frac{\partial f}{\partial x} = \frac{f(x) - f(x-\Delta x)}{\Delta x} + O(\Delta x) \tag{15}$$

The second-order accurate central difference equation can be generated by subtracting equation (15) from equation (14). This results in

$$\frac{\partial f}{\partial x} = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} + O(\Delta x)^2 \tag{16}$$

It should be noted that by ignoring the higher-order portions of the equation, error is introduced into the solution. The numerical accuracy of the solution is determined by the level at which the equations are truncated. For example, a second-order approximation is more accurate than a first-order approximation, but the second-order approximation forces the introduction of more points into the solution which increases the execution time.

3.3 Roe Scheme

The Roe scheme is based on Godunov's method in which the solution is considered as piecewise constant over each mesh cell at a fixed time. The evolution of the flow to the next time step results from the wave interactions originating at the boundaries between adjacent cells. Figure 4 shows the computational stencil and grid labeling convention.

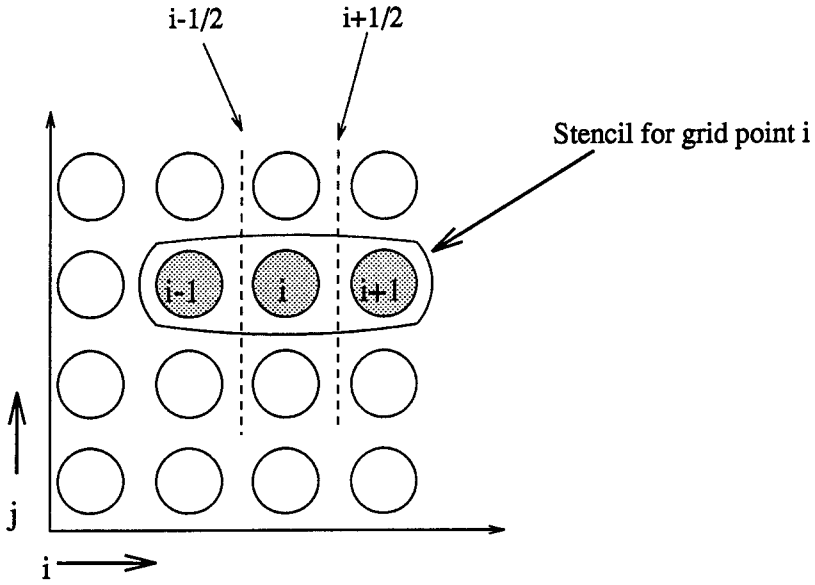


Figure 4. Computational Stencil for Roe scheme

From the governing equations for two-dimensional flow (1), the discretization using the first-order accurate, conditionally stable, explicit, upwind Roe scheme is

For the ξ sweep:

$$U^*_{i,j} = \tilde{U}_{i,j} + \frac{\lambda}{2}(D_{i+\frac{1}{2},j} - D_{i-\frac{1}{2},j}) - \frac{\lambda}{2}[y_{\eta(i,j)}(E_{i+1,j} - E_{i-1,j}) - x_{\eta(i,j)}(F_{i+1,j} - F_{i-1,j})] \quad (17)$$

where

$$D_{i+\frac{1}{2}} = R|\lambda|R^{-1}(\tilde{U}^n_{i+1,j} - \tilde{U}^n_{i,j}),$$

$$\tilde{U} = \tilde{J}U,$$

λ is the time step,

and U is the vector of conserved variables in (1),R and R^{-1} are the right and left inverses developed from flux splitting, and $|\Lambda|$ is the eigenvalue matrix.

For the η sweep:

$$U^*_{i,j} = \tilde{U}^*_{i,j} + \frac{\lambda}{2}(D_{i,j+\frac{1}{2}} - D_{i,j-\frac{1}{2}}) - \frac{\lambda}{2}[y_{\eta(i,j)}(E_{i,j+1} - E_{i,j-1}) - x_{\eta(i,j)}(F_{i,j+1} - F_{i,j-1})] \quad (18)$$

where

$$D_{i,j+\frac{1}{2}} = R|\lambda|R^{-1}(\tilde{U}^n_{i,j+1} - \tilde{U}^n_{i,j})$$

The psuedocode and time complexity analysis for the serial Roe scheme are provided in Figures 5 and 6 respectively.

PROGRAM Roe 2-D
PROGRAM INPUTS: initial values for flow solution to include number of grid points, Mach number, and convergence criteria.

1. *Initialize the flow variables.*
2. *Compute metrics of transformation.*
3. *Loop until convergence criteria is met*
4. *Compute global minimum time step based on each cell eigenvalue.*
5. *Compute fluxes at each cell interface in ξ direction.*
6. *Enforce the boundary conditions.*
7. *Compute fluxes at each cell interface in η direction.*
8. *Enforce the boundary conditions.*
9. *Update solution vector at the new time step.*
10. *End time increment.*

Figure 5. Pseudocode for Serial 2-D Roe Scheme

PROGRAM Roe 2-D
PROGRAM INPUTS: initial values for flow solution to include number of grid points, Mach number, and convergence criteria.

1. *Initialize the flow variables.* Initialize all flow variables, matrices, and grid to initial conditions. Initialization of the n elements in the matrices and grids require $O(n)$ time.
2. *Compute metrics of transformation.* Transform the physical grid into the computational grid. Mapping the two-dimensional grid requires $O(n)$ time.
3. *Loop* Continue until maximum residual is less than required convergence criteria.
4. *Compute global minimum time step on each cell eigenvalue.* The maximum time step allowable is calculated for the entire solution space. Sweeping through the two-dimensional grid requires $O(i * n)$ time.
5. *Compute fluxes at each cell interface in ξ direction.* Solve (17) in ξ plane. This requires $O(i * n)$ time.
6. *Enforce the boundary conditions.* Enforcement requires $O(1)$ time per iteration. Thus, the entire solution requires $O(i)$ time.
7. *Compute fluxes at each cell interface in η direction.* Sweep through η plane and solves (17). This requires $O(i * n)$ time.
8. *Enforce the boundary conditions.* This requires $O(i)$ time.
9. *Update solution vector at the new time step.* Compute new solution vector over the entire grid. This requires $O(i * n)$ time.
10. *End time increment.*

Figure 6. Time Complexity Analysis of 2-D Roe Scheme

The computational run time is dominated by the $O(i*n)$ time within the loop, where i is the number of iterations performed until the solution converges. Assuming that i is small in comparison to n leaves a time complexity for the Roe scheme of $O(n)$.

3.4 Beam-Warming scheme

The Beam-Warming method is widely known and utilized in CFD codes from NASA Ames (ARC2D and ARC3D). The Beam-Warming algorithm is derived from the Euler equations.

In order to solve the problem computationally, it is necessary to discretize both the computational domain and the governing equation. A solution is produced by applying the descritized governing equation to each point in the computational domain. The discretization of (1) was accomplished using a second-order spatial accurate, first-order time accurate, unconditionally stable, implicit Beam-Warming algorithm.

Application of the Beam-Warming algorithm to (1) results in the solution at each grid point i at time level $n+1$

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{2\Delta x}(U_{i+1}^{n+1} - U_{i-1}^{n+1}) \tag{19}$$

The solution dependency and solution accuracy directly determine the size of the computational stencil required. Solution of (19) requires a three-point computational stencil which yields a computational solution that is second-order accurate in space and first-order accurate in time. The computational stencil is shown in Figure 7.

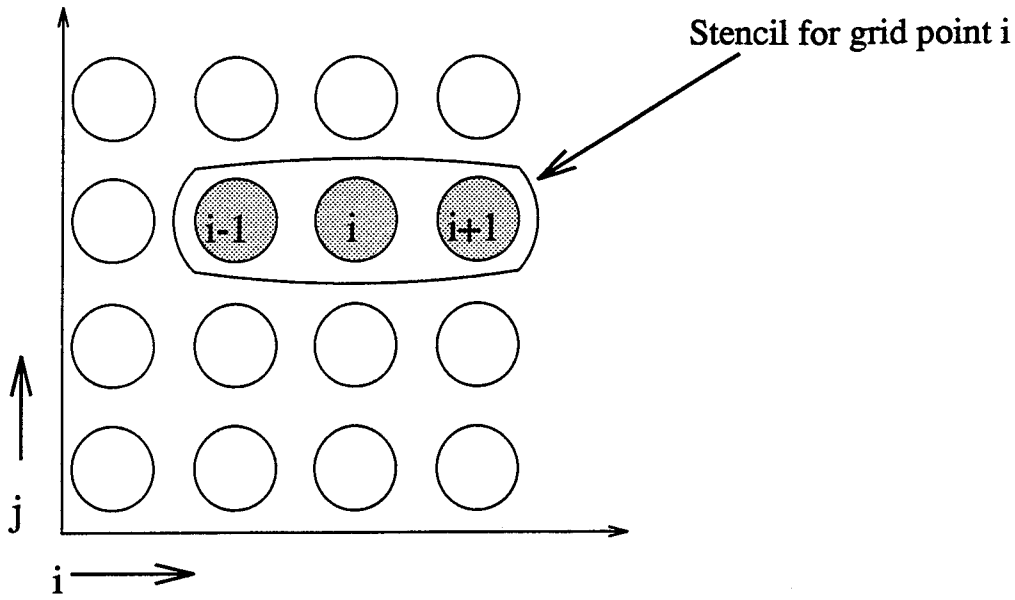


Figure 7. Computational Stencil for Beam-Warming Algorithm

Figures 8 and 9 show the psuedocode and time complexity analysis respectively for serial implementation of the Beam-Warming algorithm.

The computational run time is dominated by the $O(i * n^3)$ time to solve $x = A^{-1}b$, where i is the number of iterations performed until the solution converges. Assuming that i is small in comparison to n, the time complexity for the Beam-Warming algorithm using the Gauss-Jordan method is $O(n^3)$.

In order to decrease the complexity, several other schemes that solve $x = A^{-1}b$ have been developed. The more commonly used algorithms are the ADI, Gauss-Seidel, Red-Black Gauss Seidel, SOR, Jacobi, and LSOR methods. For example, solving the tridiagonal system using ADI can reduce the solution time complexity to $O(n^2)$.

Thus, substituting the ADI method of solving the linear system of equations reduces the overall time complexity to $O(n^2)$.

PROGRAM Beam-Warming 2-D
PROGRAM INPUTS: initial values for flow solution to include number of grid points, Mach number, and convergence criteria.

1. *Initialize the flow variables.*
2. *Compute metrics of transformation.*
3. *Compute local time step.*
4. *Loop until convergence criteria is met*
5. *Sweep in η direction.*
6. *Sweep in ξ direction.*
7. *Solve $x = A^{-1}b$.*
8. *Apply boundary conditions.*
9. *End time increment.*

Figure 8. Pseudocode for Beam-Warming Algorithm

3.5 Grid Generation

The grid for the cylinder was generated using a FORTRAN routine called GEOM (6). This subroutine generates a grid by calling GEOM for the symmetry plane and outflow radii and then interpolating for all points between these two lines. Figure 10 shows the resultant grid.

3.6 Boundary Conditions

The boundary conditions that were implemented for flow over a cylinder are described below. In each case, the free-stream flow is enforced with the following conditions

$$\begin{aligned} \rho^* &= 1 \\ \rho^* u^* &= 1 \\ \rho^* v^* &= 0 \end{aligned} \tag{20}$$

PROGRAM Beam-Warming 2-D
PROGRAM INPUTS: initial values for flow solution to include number of grid points, Mach number, and convergence criteria.

1. *Initialize the flow variables.* Initialize all flow variables, matrices, and grid to initial conditions. Initialization of the n elements in the matrices and grids require $O(n)$ time.
2. *Compute metrics of transformation.* Transform the physical grid into the computational grid. Mapping the two-dimensional grid requires $O(n)$ time.
3. *Compute local time step.* Calculate the maximum local time step allowable for each grid point. Sweeping through the two-dimensional grid requires $O(n)$ time.
4. *Loop* Continue until maximum residual is less than required convergence criteria.
5. *Sweep in η direction.* Solve (19) in η plane, computing right-hand side (RHS) terms for η direction. This requires $O(i * n)$ time.
6. *Sweep in ξ direction.* Sweep through ξ plane and solve (19) in ξ plane, computing RHS terms for ξ direction. This requires $O(i * n)$ time.
7. *Solve $x = A^{-1}b$.* Factor and back-substitute to get A^{-1} and perform a vector-matrix multiplication. The standard Gauss-Jordan technique requires $O(i*n^3)$ time to invert the A matrix (31) and dominates the time to perform vector-matrix multiplication.
8. *Update Boundary Conditions* Apply boundary conditions. This requires $O(i * 1)$ time.
9. *End time increment.*

Figure 9. Time Complexity Analysis of Beam-Warming Algorithm

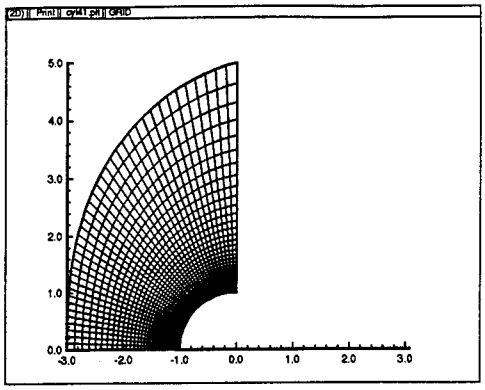


Figure 10. Grid for cylinder

Flow Over a Cylinder

To take advantage of the symmetry inherent in this geometry, a symmetry plane is passed through the center of the cylinder. The flow conditions can then be calculated for the flow over the "top" of the cylinder with the flow over the "bottom" being the mirror image. Ghost points are needed in the vicinity of the upstream symmetry line and within the cylinder in order to properly impose the boundary conditions. The boundary conditions for the cylinder surface are

$$\begin{aligned} \rho^* u^* &= 0 \\ \rho^* v^* &= 0 \\ \frac{\partial T^*}{\partial n} &= 0 \\ \frac{\partial p^*}{\partial n} &= 0 \end{aligned} \tag{21}$$

For the ghost points below the symmetry line, the following boundary conditions are enforced

$$\begin{aligned} \rho^* u^* &= 0 \\ \rho^* v^* &= -\rho^* v^* \\ \frac{\partial T^*}{\partial n} &= 0 \\ \frac{\partial p^*}{\partial n} &= 0 \end{aligned} \tag{22}$$

3.7 Summary

The governing equations, pseudocode, time-complexity analysis, and boundary conditions for the serial implementation of the Roe Scheme and Beam-Warming algorithms are presented. The implementation issues and a parallel performance analysis is presented in Chapter 4.

IV. Parallel Implementation

This chapter presents the parallel computational design process and a parallel time-complexity analysis which were used to determine the expected parallel run time for the algorithms chosen. Both explicit and implicit algorithms are explored due to the differences in the methods required to parallelize them. While the reader is assumed to be familiar with explicit and implicit schemes, a comparison of the two algorithms is shown in Figure 11.

	Explicit Methods	Implicit Methods
Solution Procedure	point-wise	must solve Ax=b
Allowable Time Step (stability)	small	unlimited
Number of time steps to converge	typically large	can be small
Computational work per timestep	small	large
Parallelizability	easy	can be difficult due to data dependencies

Figure 11. A Comparison of Explicit and Implicit Methods (4)

The Roe explicit scheme was easily parallelized because no data dependencies exist, i.e. all elements of the solution vector at t_{n+1} only depend on the solution at t_n. The main disadvantage of an explicit scheme is that only small, fixed time steps are allowed. The Beam-Warming implicit scheme is the more difficult algorithm to parallelize because data dependencies exist. Additionally, solving $Ax = b$ inefficiently can easily eliminate the potential gain of taking large time steps.

The governing equations (1), boundary conditions (20) (21)(22), pseudocode (Figures 5 and 8), and time-complexity analysis (Figures 6 and 9) for the serial methods have already

been presented. The proposed methodology to solving the above problem was to parallelize the serial algorithms by developing efficient yet generic routines, perform a complexity analysis on the parallel pseudocode, and derive the general performance characteristics of the algorithms. Using the parallel measurement methods from (23) and (22), the test matrices were then constructed.

4.1 Roe Scheme

Development of a good parallel algorithm should describe the solution of a problem without taking into account implementation. UNITY (Unbounded Nondeterministic Iterative Transformations) is a parallel program specification and design language that provides a description of what needs to be accomplished without regard to how it should be accomplished (9). A UNITY program should attempt to express the maximum parallelism possible. The operations separated by parallel lines (||)can only be performed synchronously. Those separated by a box ([]) can be done asynchronously.

Figure 12 reflects the top-level design of the Roe scheme. In general, the algorithm sweeps in one dimension and evaluates the terms $|\Lambda|$,R, and R^{-1} for t_{n+1} using data from t_n. The amount of data required from neighboring grid points is based on the numerical accuracy of the solution. The solution vector U^{n+1} is updated and the algorithm then sweeps in the other dimension. U^{n+1} is again updated and replaces the data stored in U^n. The time step is then incremented and the sweeps repeated until the residual falls below some convergence criteria specified within the algorithm. Formal definitions for $|\Lambda|$,R, and R^{-1} can be found in (6).

Program *RoeScheme*
declare

U	:array$[\rho, U, V, E_t]$
X	:array$[I, J]$ of integer
Y	:array$[I, J]$ of integer
IDM	: integer
JDM	: integer
R	: real
R^{-1}	: real
Λ	: real
$N\_ITERATION$	: integer
$MAX\_ITERATION$	: integer
$CONVERGENCE$	: integer
$RESIDUAL$	: integer

initially

η_SWEEP	= false
ξ_SWEEP	= false
$N\_ITERATION$	$= 0$
$MAX\_ITERATION$	$= \infty$
$RESIDUAL$	$= \infty$

assign

$\langle$

$\langle\langle\langle R_\eta = f\,(U^n_{\eta_x,\eta_y})\rangle[]\langle R^{-1}_\eta = f\,(U^n_{\eta_x,\eta_y})\rangle[]\langle\Lambda_\eta = f\,(U^n_{\eta_x,\eta_y})\rangle$
if$(0 \le N_ITERATION \le MAX_ITERATION \wedge \neg\xi_SWEEP \wedge \neg CONVERGENCE)\rangle$

||

$\langle\langle R_\xi = f\,(U^n_{\xi_x,\xi_y})\rangle[]\langle R^{-1}_\xi = f\,(U^n_{\xi_x,\xi_y})\rangle[]\langle\Lambda_\xi = f\,(U^n_{\xi_x,\xi_y})\rangle$
if$(0 \le N_ITERATION \le MAX_ITERATION \wedge \neg\eta_SWEEP \wedge \neg CONVERGENCE)\rangle\rangle$

||

$\langle\langle U^{n+1}_{\xi,\eta} = f\,(R_\eta, R^{-1}_\eta, \Lambda_\eta, R_\xi, R^{-1}_\xi, \Lambda_\xi)\rangle$
if$(0 \le N_ITERATION \le MAX_ITERATION \wedge$
$\neg\eta\_SWEEP \wedge \neg\xi\_SWEEP \neg CONVERGENCE)\rangle\rangle$

end

Figure 12. UNITY Description of the Roe Scheme

The invariant, fixed point, and progress conditions for the Roe scheme are

invariant
$0 \leq N\_ITERATION \leq MAX\_ITERATION$
FP $\equiv$
$N\_ITERATION = MAX\_ITERATION \vee$
$\forall_{i,j} :: residual_{i,j} < CONVERGENCE \wedge N\_ITERATION < MAX\_ITERATIONS$
progress
$\neg$ **FP** $\wedge N\_ITERATION = k \mapsto N\_ITERATION = k + 1$

Based on the UNITY description, it is evident that the explicit scheme has fewer data dependencies than the implicit scheme and is inherently more parallel. With few modifications, the explicit Roe scheme can be modified to efficiently run on multiple processors. These modifications are presented next.

4.1.1 Roe Scheme Implementation Details. Parallelization of the serial Roe scheme required implementation of the domain decomposition, load balancing, and buffer exchange algorithms. Additionally, the Roe scheme required implementation of global routines to determine the maximum and minimum values spread across the processors.

Domain Decomposition

As mentioned previously, there are several methods to decompose the domain. Because the grids generated are not unstructured or dynamic in nature, decomposition techniques such as RSB, RGB, and RCB were not used. Instead, the overlapping techniques presented in (8) were considered. The two techniques are

- Non-overlapped decomposition and
- Overlapped decomposition.

The non-overlapped technique involves decomposition of the domain into separate blocks with no space for copies of data from adjacent processors. In the overlapped technique, a number of "buffers" are maintained along the edges of each processors computational block. These techniques applied to a 2-D problem are shown in Figure 13. With the

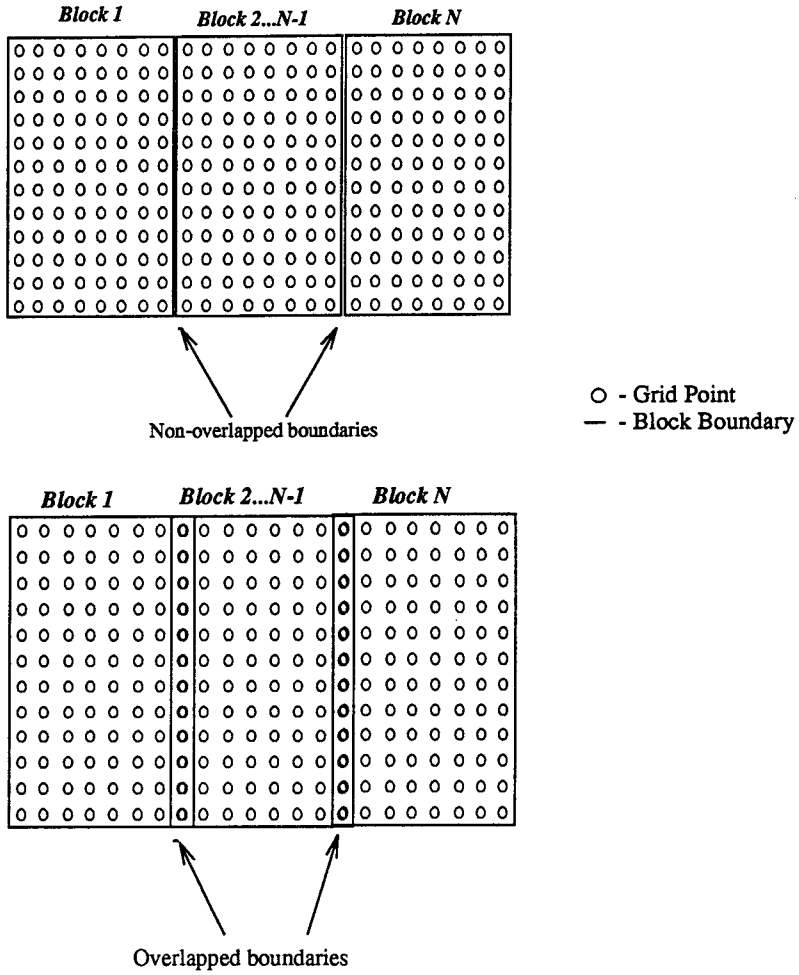


Figure 13. Non-overlapped and Overlapped Domains for a 2-D Problem

overlapped technique, the width of the buffer is determined by the computational stencil used. For example, an algorithm of second-order accuracy using a three-point stencil in each dimension would require a buffer only one grid point wide; a fourth-order accurate solution requiring a five-point stencil in each dimension would require a buffer two grid

points wide. Figure 14 shows this relationship applied to a one-dimensional partitioning scheme.

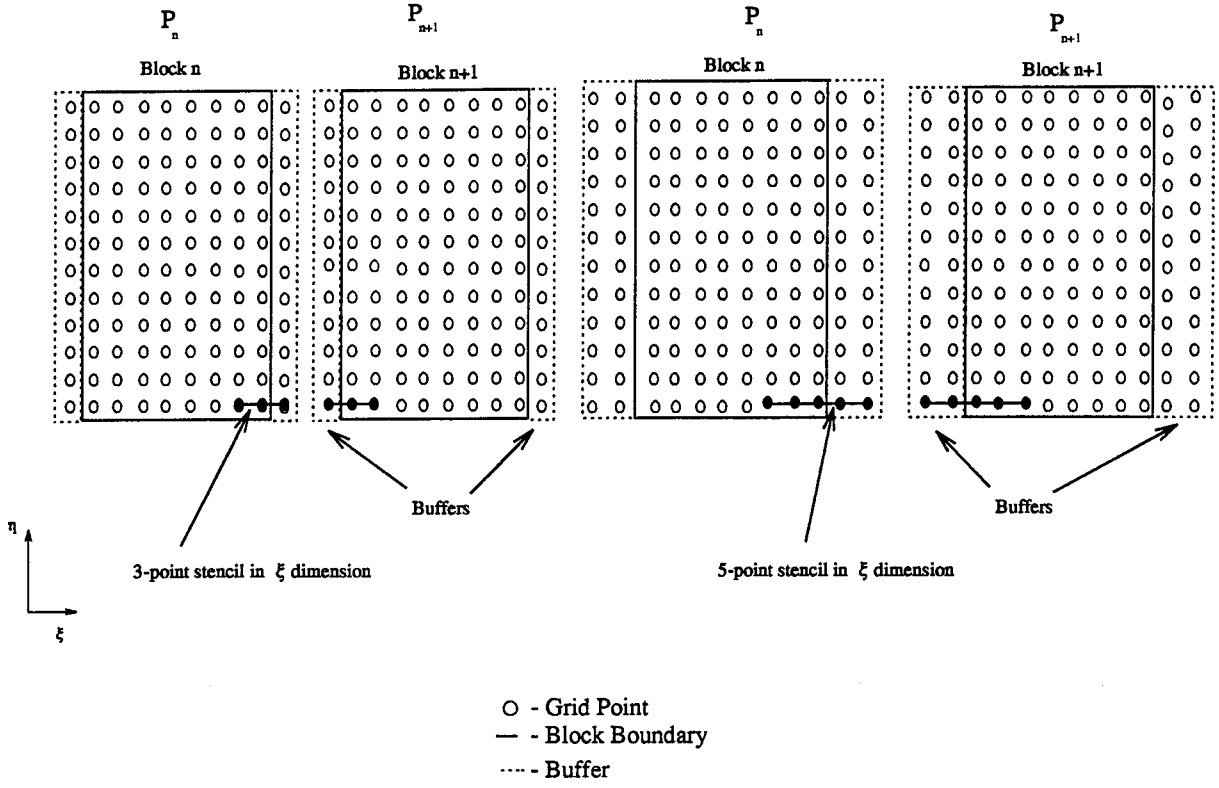


Figure 14. Example of Buffer Widths

Decomposition without overlapping requires no additional total memory to store the entire domain across p processors. However, communication is greatly affected because message traffic is greatly increased. This is because each element transferred between two nearest neighbor processors incurs the communication startup time, the most expensive part of communication. Additionally, this communication restricts the inherent parallelism of the algorithm by requiring communication to be performed before each grid point can perform its calculations.

While overlapped decomposition requires additional amounts of memory to store buffer arrays, it greatly reduces the communication overhead by packing an entire buffer into one message and incurring only one message startup cost. After the buffer arrays are updated, each processor is able to process independently of its neighbor processors until the next synchronization point.

Decomposition of the computational domain has a direct bearing on the communications efficiency of a program. The total communication time to send a message is given by

$$t_{comm} = t_s + lt_h + mt_w \tag{23}$$

where t_s is the startup time required to handle a message at the sending processor, t_h is the time to reach the next processor, l is the number of "hops" between sending and receiving processor, m is the size of the message in bytes, and t_w is the per-word transfer time (23). For the Intel Paragon, t_s dominates the other parameters so significantly that the time to transfer a message between nearest neighbors is approximately the same as between the two furthest nodes. Thus, (23) reduces to

$$t_{comm} = t_s \tag{24}$$

Applying (23) and (24) to an example where a buffer one point in width (each containing eight bytes of data) is exchanged along a one-dimensional boundary of length 21 would result in

$$T_{comm} = 21 * (t_s + lt_h + 8 * t_w)$$

or

$$T_{comm} = 21 * t_s$$

on the Paragon. Exchanging the entire buffer in one message would result in a communication time of

$$T_{comm} = 1 * (t_s + lt_h + 168 * t_w)$$

or

$$T_{comm} = t_s$$

on the Paragon. Thus, when memory is not a limitation,exchanging an entire data buffer results in the more efficient method of updating adjacent points.

For this implementation of the Roe Scheme, the domain was decomposed into strips in the ξ-dimension. To accommodate the 3-point stencil shown in figure 4, a buffer one point in width was used. This decomposition is shown in Figure 15. Note that buffers are not needed along the outside edges of Block 0 and Block N.

After decomposing the domain, each processor's starting and ending indices must be adjusted. Solving for a time step involves looping from $ISP1$ to $IEM1$ in the ξ direction and $JSP1$ to $JEM1$ in the η direction and then imposing the boundary conditions along IS, IE, JS, and JE. For a parallel implementation, maintaining these loop parameters across all processors would result in an incorrect solution because the boundaries are spread across many processors. As a result, depending on the *blocktype* contained on a processor,

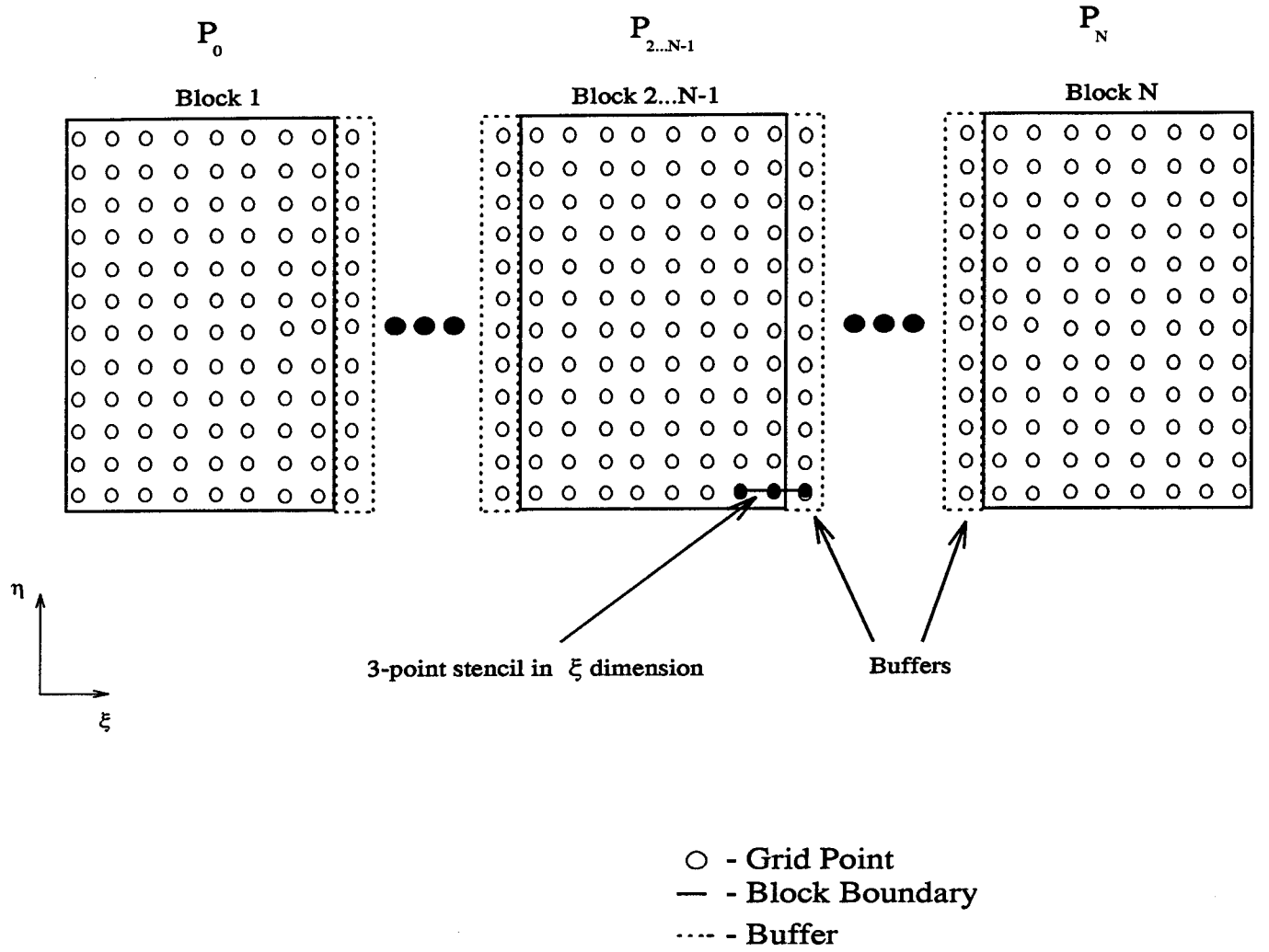


Figure 15. 1-D Decomposition of Computational Domain

the loop indices $ISP1, IEM1$ are adjusted at initialization as follows:

$$ISP1 = \begin{cases} IS+1 & \text{if } blocktype = -1 \\ IS & \text{otherwise} \end{cases}$$

$$IEM1 = \begin{cases} IE-1 & \text{if } blocktype = 1 \\ IE & \text{otherwise} \end{cases}$$

Because the problem was partitioned in one dimension, the boundaries in the η dimension were distributed across all processors. Thus, no special consideration was required for $JSP1$ and $JEM1$. Figure 16 shows this adjustment.

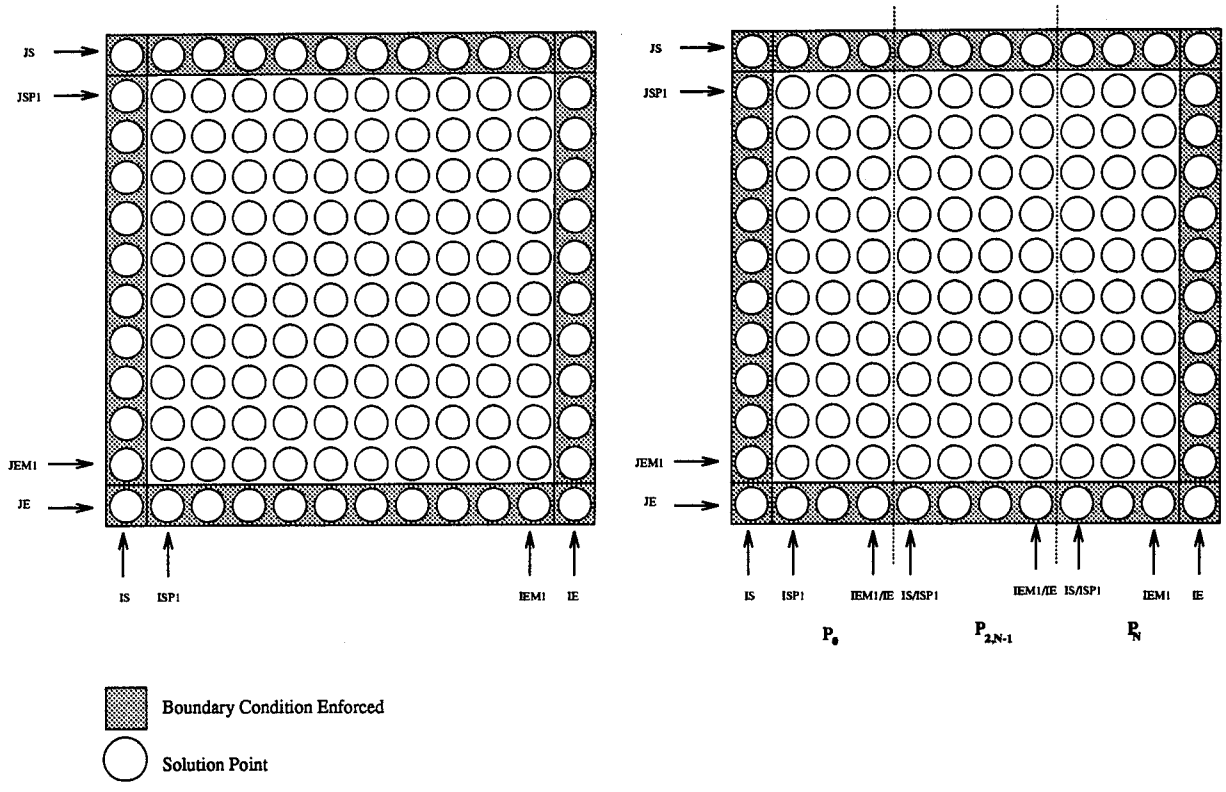


Figure 16. Loop Index Adjustment Required for Parallel Implementation

Load Balancing

An efficient implementation of a parallel program results in each processor performing meaningful computational work as much as possible. Therefore, the workload must be evenly distributed among the processors while taking into account specialized processing requirements of particular processors such as grid generation, data distribution, and boundary condition enforcement.

This implementation used a one-dimensional partitioning algorithm, *PARTITION1D*, which evenly divides the computational domain across n processors. If the computational domain does not evenly divide among the processors, the algorithm initially distributes equal amounts of the computational domain to each processor, calculates the undistributed workload, and readjusts the indices on each processor to account for an uneven workload.

Using *PARTITION1D*, no processor ever has more than JDM additional points than any other processor, where JDM is the length of the partition in the η dimension.

Specialized processing is required for several tasks including:

- Data Initialization,
- File I/O,
- Grid Generation,
- Boundary Condition Enforcement, and
- Exchanging Buffer Arrays.

Data initialization and grid generation can be implemented across all processors simultaneously and the cost reading an input file can be amortized over the entire time to solve the problem. By removing all specialized processing requirements (except boundary

condition enforcement and exchanging buffer arrays) from the main iterative loop, the amount of computational work performed by each processor becomes relatively similar. The only differences in computational work performed on each block then become:

1. Block 1 and Block N must enforce an additional boundary condition and
2. Blocks 2...$(N-1)$ must exchange one additional buffer.

Figure 17 shows these differences.

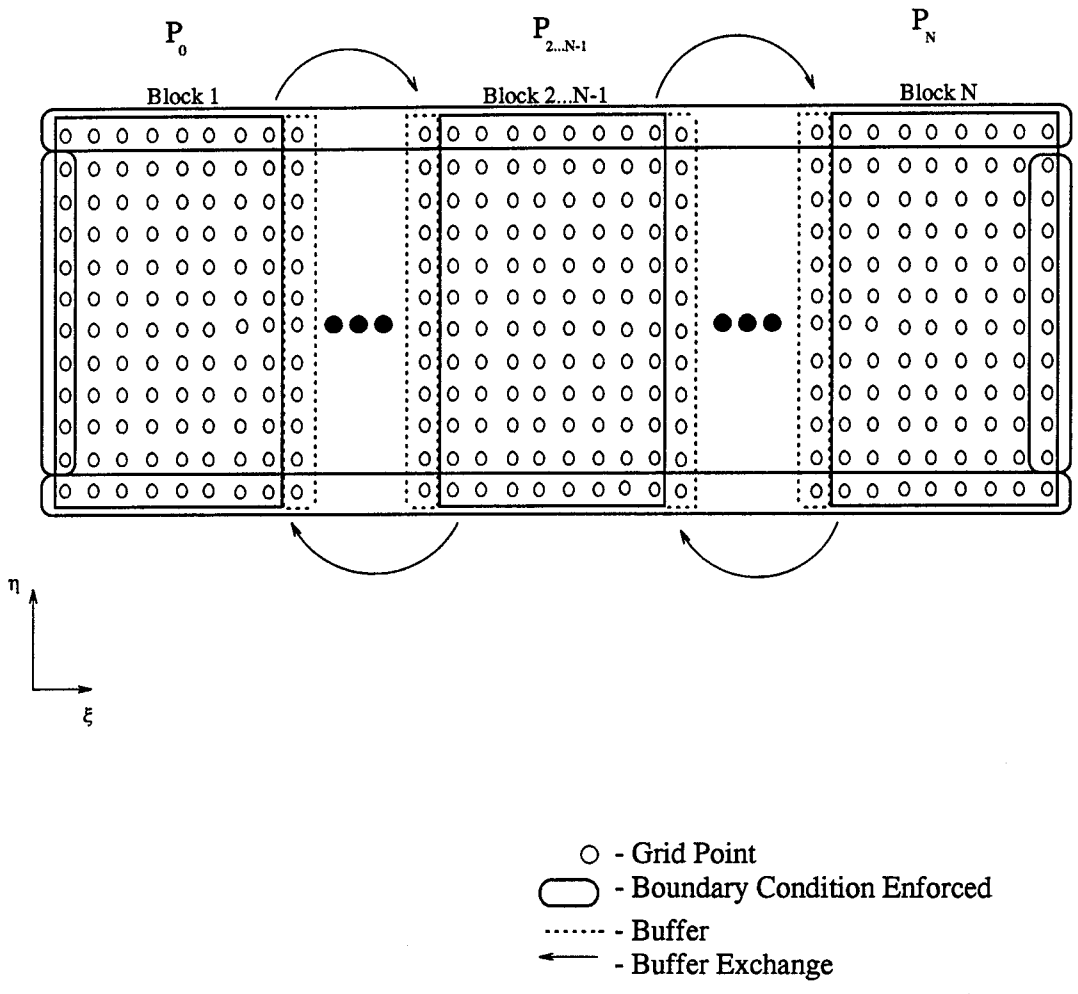


Figure 17. Differences in Computational Work Required for Each Blocktype

See Appendix A for the code listing and explanation of the algorithm.

Buffer Exchange

As mentioned above, some method for exchanging buffer arrays is required. Before any messages are passed between processors, each block must be assigned a "blocktype" which defines its communication pattern as follows:

- Blocktype -1 - passes and receives messages only from the processor to its right.
- Blocktype 0 - passes and receives messages from the processor to its right and left.
- Blocktype 1 - passes and receives messages only form the processor to its left.

The message passing technique on the Paragon greatly facilitates the message passing process. The Paragon passes a message using the following format for a synchronous message *send*

$$csend(msg\_id, msg, num\_bytes, dest, pid) \tag{25}$$

where *msg_id* is a unique message identifier, *msg* is the beginning address of the data to be sent, *num_bytes* is the number of bytes to be transferred, *dest* is the destination node, and *pid* is the process identification (1). The send starts at the address given by *msg*, places the next *num_bytes* into the message, and sends it to *dest*. Figure 18 shows how a 2-D buffer is mapped by row into a message packet.

Because the domain was partitioned in the ξ-dimension, each buffer contained a *column*of data. Because data is mapped by *row* into a message packet, the column of data must be disassembled and mapped to a row of data, sent to *dest*, and reassembled into a column of data. The routine *EXCHANGEBUFFS* automatically performs the disassemble-send-reassemble process. Using the blocktype convention explained above, a

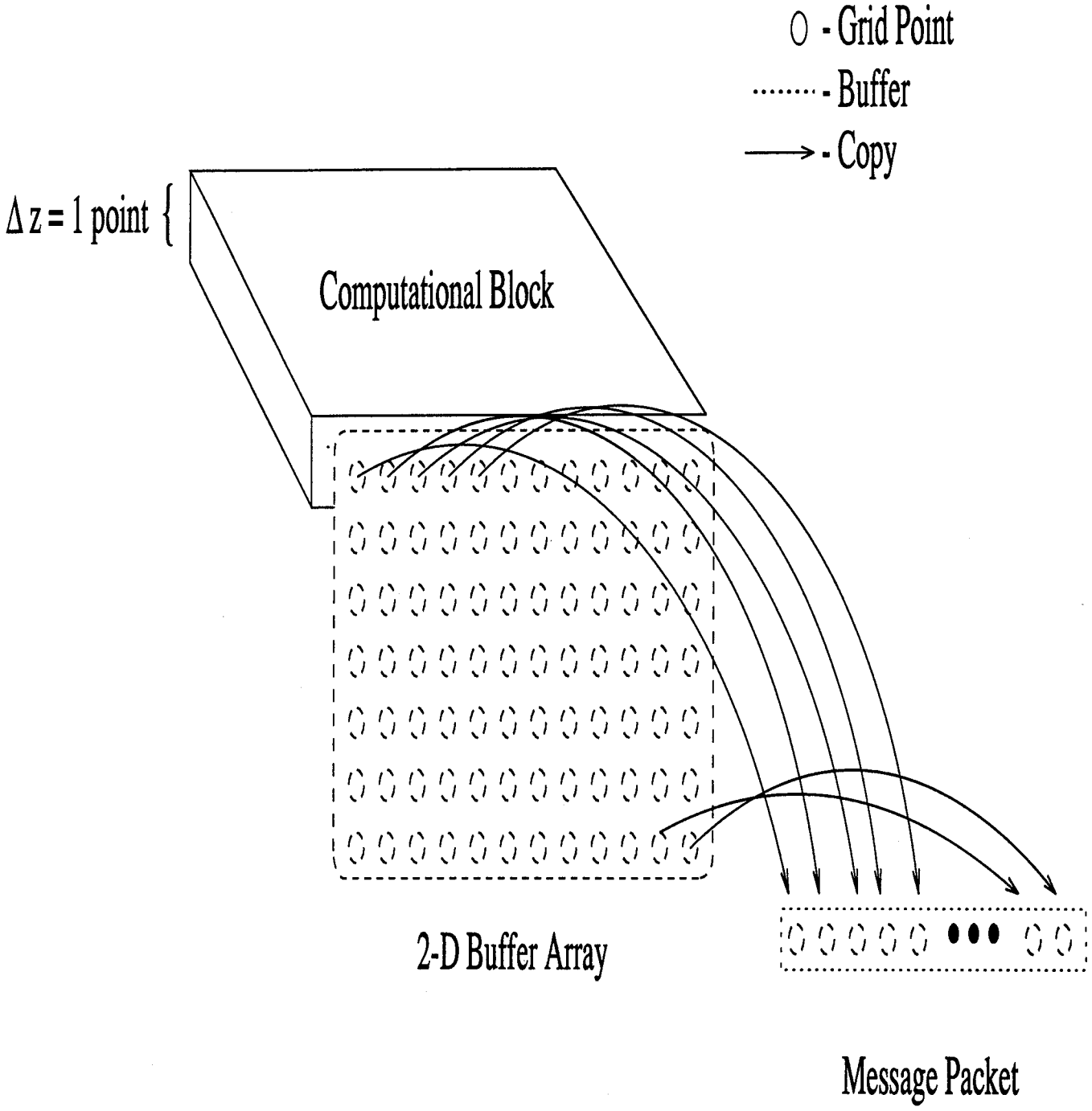


Figure 18. Mapping a 2-D Buffer Array Row into a Message Packet

call to *EXCHANGEBUFFS* followed by the array to be exchanged automatically exchanges buffer data between processors. Figure 19 shows this process.

EXCHANGEBUFFS was also used for exchanges of buffers extending in three dimensions by adding a "mode" as one of the input parameters. A conditional checks the "mode" of the buffer exchange which determines whether or not the data structure is swept in the third dimension. The data exchange is then carried out as described previously. See Appendix A for the code listing and explanation of the associated algorithm.

Global Routines. Use of the Intel Paragon provided global routines generates the most efficient global communications due to dynamic algorithm selection. This algorithm considers several ways of exchanging information between nodes and selects the one that minimizes the time to perform the global operation (1).

The Roe scheme requires that for each iteration, a minimum time step be calculated based on the largest eigenvalue of the system. By executing the global routine *gdlow(timestep)*, all timesteps are collected and the lowest timestep is stored on each processor. This routine is a *synchronizing call* and blocks the current processor from continuing until all other processors have issued the same call.

Another global routine, *gdhigh(max_norm)*, is required to determine the maximum norm located in the system. Each processor sweeps through $\frac{n}{p}$ elements and determines the largest norm on each processor. The p processors then call the global routine *gdhigh(maxnorm)*, and a global maximum is computed and stored on each processor.

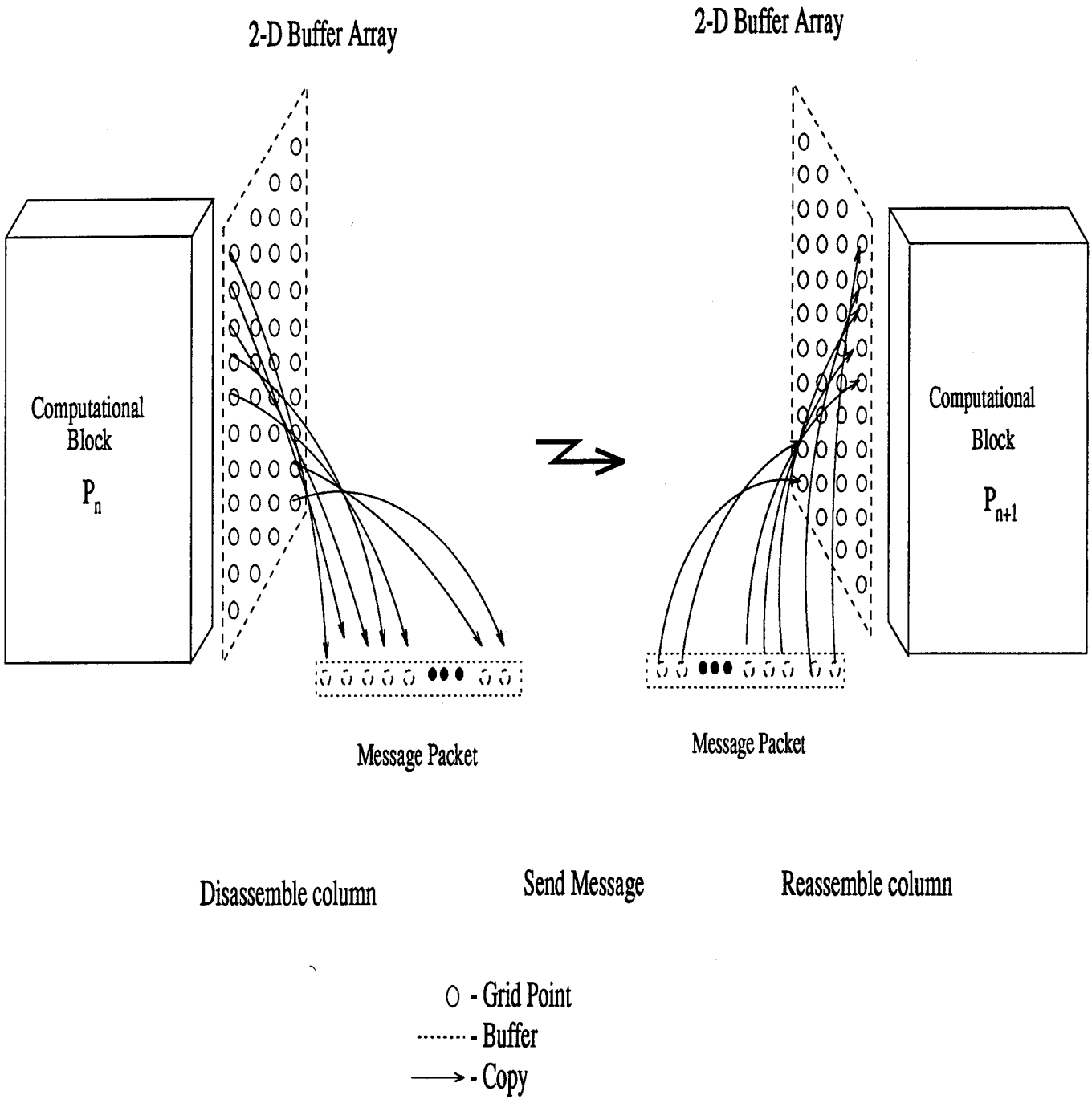


Figure 19. Mapping a 2-D Buffer Array Column into a Message Packet

PROGRAM Roe 2-D
PROGRAM INPUTS: initial values for flow solution to include number of grid points, Mach number, and convergence criteria.

1. *Initialize the flow variables.* All flow variables, matrices, and grid are initialized to initial conditions. Initialization of the n elements in the matrices and grids require $O(\frac{n}{p})$ time.
2. *Compute metrics of transformation.* The physical grid is transformed into the computational grid. Mapping the two-dimensional grid requires $O(\frac{n}{p})$ time.
3. *Exchange initial vectors between buffer planes.* This step updates the boundary information on each processor. This requires $O(\sqrt{n})$ time.
4. *Loop* Continue until maximum residual is less than required convergence criteria or maximum iterations reached.
5. *Compute global minimum time step on each cell eigenvalue.* The maximum time step allowable is calculated for the entire solution space. Sweeping through the two-dimensional grid requires $O(\frac{i*n}{p})$ time.
6. *Perform Global Reduction* This step requires that a global reduction across p processors be performed to get the minimum time step that the solution can advanced. A global reduction can be performed in $\log p$ time with a communication overhead of $\sqrt{p}$. Total time required is $O(i * \sqrt{p})$.
7. *Compute fluxes at each cell interface in ξ direction.* Solves (17) in ξ plane. This requires $O(\frac{i*n}{p})$ time.
8. *Compute fluxes at each cell interface in η.direction* Sweeps through η plane and solves 17 in η plane. This requires $O(\frac{i*n}{p})$ time.
9. *Update solution vector at the new time step.* Computes new solution vector over the entire grid, requiring $O(\frac{i*n}{p})$ time.
10. *Exchange the new solution vector between buffer planes.* This step updates the boundary information on each processor. This requires $O(i * \sqrt{n})$ time.
11. *End time increment.*

Figure 20. Pseudocode/Time Complexity for Parallel 2-D Roe Scheme

4.1.2 Parallel Roe Scheme Pseudocode/Time Complexity Analysis. Figure 20 shows the pseudocode and time complexity of the parallel implementation of the Roe scheme. The total time required to run the algorithm is the sum of the steps in the pseudocode. The computational work is performed in steps 1,2,5,7,8, and 9. Steps 3,6, and 10 represent overhead due to communication. For large i, steps 1 and 2 be incorporated into the time to perform step 5. Because the time to communicate between any two processors on the Paragon is constant, the $\sqrt{n}$ term dominates the communications overhead. Thus, steps 3 and 6 can be incorporated into step 10 and reduced to $O(i * \sqrt{n})$. Thus, the total time required to solve the problem in parallel is

$$T_p = \overbrace{\Theta\left(\frac{i*n}{p}\right)}^{computation} + \overbrace{\Theta\left(i*\sqrt{n}\right)}^{communication}$$

Thus,

$$T_o = \Theta\left(i*\sqrt{n}\right)$$

$$W = \Theta\left(\frac{i*n}{p}\right)$$

where T_o and W are the overhead and useful work performed respectively.

Speedup measures the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors.

$$S = \frac{T_1}{T_n + T_o}$$

where T_1 and T_n are the time taken for solving a problem on one and n processors respectively and T_o the amount of time consumed by overhead.

The theoretical speedup can be predicted by dividing the complexity of the serial implementation by the complexity of the parallel implementation. The theoretical speedup for this implementation of the Roe scheme is

$$S = \frac{\Theta(i * n)}{\Theta(\frac{i*n}{p}) + \Theta(i * \sqrt{n})}$$

Efficiency is a measure of the fraction of time for which a processor is usefully utilized and is defined as the ratio of the speedup to the number of processors. Ideally, speedup is equal to p and the efficiency is equal to one. Thus, the equation for efficiency is

$$E = \frac{S}{p}$$

The projected efficiency for the Roe scheme algorithm is

$$E_{rs} = \frac{\frac{\Theta(i*n)}{\Theta(\frac{i*n}{p})+\Theta(i*\sqrt{n})}}{p} = i\left(\frac{n*p}{n+\sqrt{n}}\right)$$

The isoefficiency function characterizes the amount of parallelism inherent in a parallel algorithm (23). The isoefficiency function for an algorithm can be computed by setting the dominant term in the overhead equal to the computational work. For the 2D Roe scheme the isoefficiency function is calculated to be

$$n = p^2$$

Thus, as the number of processors are increased by p^2, the size of the problem must be increased by n in order to provide the same efficiency.

The communications overhead dominates the computational part of the algorithm. Since the serial algorithm is $\Theta(i * n)$, the processor/time product is optimal for $n > p^2$.

Figure 21 shows the parallel flow of the Roe scheme.

4.2 Parallel Algorithm Development: Beam-Warming

This section concentrates on presentation a high-level UNITY design for the Beam-Warming algorithm. The UNITY description will then be used to map the serial algorithm to an efficient parallel algorithm that exploits the inherent parallelism of the algorithm to the fullest extent.

Figure 22 reflects the top-level design of the Beam-Warming algorithm. In general, the algorithm sweeps in one dimension at a time with each point in the grid performing calculations at t_{n+1} based on data from t_n and t_{n+1} time steps. The calculations and number of neighboring grid points required are dependent on the method used and numerical accuracy of the solution. Each grid point generates the state of the grid point at time t_{n+1} and a set of simultaneous equations are solved. A residual is then calculated at each grid point. This is repeated until the residual falls below some convergence criteria specified within the algorithm.

The invariant is derived from the initial state for $N\_ITERATION$ and monotonically increments $N\_ITERATION$ until the fixed point is reached. The fixed point states that one of two conditions must occur. Either the application reaches a maximum number

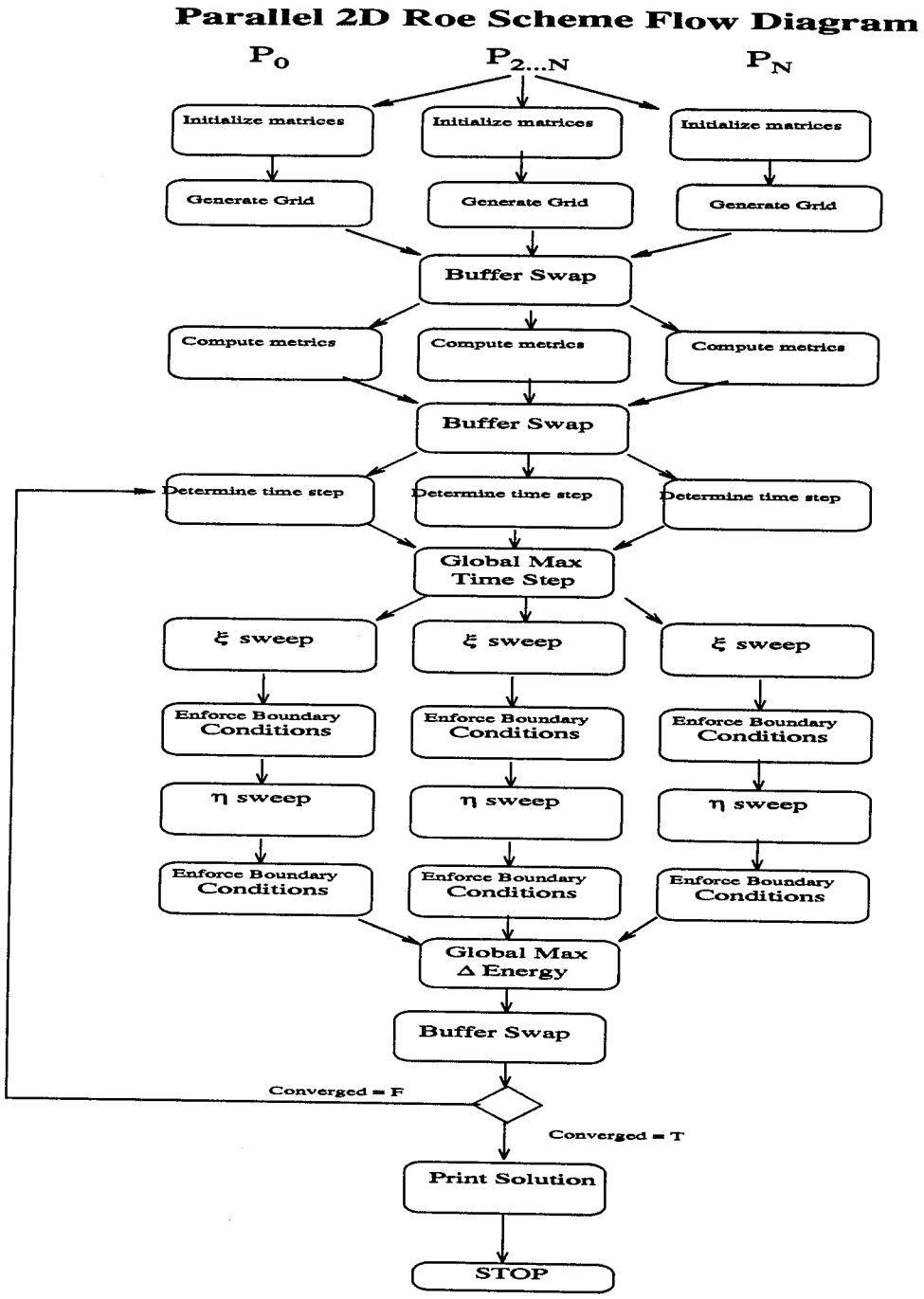


Figure 21. Parallel Flow of Roe Scheme Code.

Program $Beam - Warming$
declare

U	:array$[\rho, U, V, E_t]$
X	:array$[I, J]$ of integer
Y	:array$[I, J]$ of integer
IDM	: integer
JDM	: integer
$N\_ITERATION$	: integer
$MAX\_ITERATION$	: integer
$CONVERGENCE$	: integer
$RESIDUAL$	: integer

initially

η_SWEEP	= false
ξ_SWEEP	= false
$N\_ITERATION$	$= 0$
$MAX\_ITERATION$	$= \infty$
$RESIDUAL$	$= \infty$

assign

$\langle$

$\langle\langle U_{\eta}^{n+1} = f\,(U_{\eta_x,\eta_y}^{n}, U_{\eta_x,\eta_y}^{n+1})\rangle || \langle RHS = f\,(U_{\eta_x,\eta_y}^{n}, U_{\eta_x,\eta_y}^{n+1})\rangle [] \langle A = f\,(U_{\eta_x,\eta_y}^{n}, U_{\eta_x,\eta_y}^{n+1})\rangle$
if $(0 \leq N\_ITERATION \leq MAX\_ITERATION \wedge \neg\xi\_SWEEP \wedge \neg CONVERGENCE)\rangle$

||

$\langle\langle U_{\xi}^{n+1} = f\,(U_{\xi_x,\xi_y}^{n}, U_{\xi_x,\xi_y}^{n+1})\rangle || \langle RHS = f\,(U_{\xi_x,\xi_y}^{n}, U_{\xi_x,\xi_y}^{n+1})\rangle [] \langle A = f\,(U_{\xi_x,\xi_y}^{n}, U_{\xi_x,\xi_y}^{n+1})\rangle$
if $(0 \leq N\_ITERATION \leq MAX\_ITERATION \wedge \neg\eta\_SWEEP \wedge \neg CONVERGENCE)\rangle\rangle$

||

$\langle\langle x = A^{-1} * RHS$
if $(0 \leq N\_ITERATION \leq MAX\_ITERATION \wedge \neg\eta\_SWEEP \wedge$
$\neg\xi\_SWEEP \neg CONVERGENCE)\rangle\rangle$

end

Figure 22. UNITY Description of the Beam-Warming Algorithm

of iterations,$MAX\_ITERATION$, or the maximum residual is less than the convergence criteria specified within the program. To show progress we note that if the program is not at a fixed point, then the number of iterations,$N\_ITERATION$, is increased by one.

invariant
$0 \leq N\_ITERATION \leq MAX\_ITERATION$
FP $\equiv$
$N\_ITERATION = MAX\_ITERATION \vee$
$\forall_{i,j} :: residual_{i,j} < CONVERGENCE \wedge$
$N\_ITERATION < MAX\_ITERATIONS$
progress
$\neg$ **FP** $\wedge N\_ITERATION = k \mapsto N\_ITERATION = k + 1$

Analysis of this UNITY description shows inherent limitations to achieving maximum parallelization of the Beam-Warming algorithm. The η and ξ sweeps cannot be performed simultaneously and the equations can only be solved simultaneously after both sweeps have been completed. The parallel efficiency of calculating $Ax = b$ must also be considered.

Based on the UNITY description, only minor modifications need to be made to the serial implementation in order to provide efficient parallel performance. These modifications are presented below.

4.2.1 Beam-Warming Implementation Details. Domain Decomposition

The domain is decomposed into overlapping computational domains with a buffer width of one grid point.

Load Balancing

As in the Roe Scheme implementation, the computational domain is divided into equal partitions using *PARTITION1D.*

Buffer Exchange

The *EXCHANGEBUFFS* routines are reused to exchange buffer data between processors.

Transposition

Transposition of matrix data was required after calculating molecular viscosity and for the solution of the block tridiagonal systems. A complete matrix requires $O(\frac{n^2}{2p})$ time to transpose (23).

Solving $\boldsymbol{Ax} = \boldsymbol{b}$

While several specialized methods exist for solving a linear system of equations, **specialized linear algebra libraries** offer the best source of efficient solution to the linear system of equations. Parallel implementations of BLAS routines such as SCALAPACK, DES, IES, and SES (1) are highly tuned linear algebra packages.

Due to the availability of parallel linear algebra libraries, the SES linear algebra solver library was used to solve the system of equations. The SES solver uses the direct method to solve the system of equations. It performs a block LU factorization with full partial pivoting and then solves for x across p processors. Because the serial implementation of the Beam-Warming algorithm is $O(n^2)$, the SES solver must be at least $O(\frac{n^2}{p})$ in order for

any benefit from parallelization to occur. It is assumed that the SES solver can perform LU decomposition and solve a system of equations on p processors in $O(n^2)$ time.

4.2.2 Parallel Beam-Warming Algorithm Pseudocode/Time Complexity Analysis. The pseudocode and time complexity analysis of the parallel Beam-Warming code is shown in Figure 23. The total time required to run the algorithm is the sum of the steps in the pseudocode. The computational work is performed in steps 1,2,4,5,7,8,9, and 11. Steps 3 and 10 represent overhead due to communication. For large i, steps 1,2,4, and 5 can be incorporated into the time to perform step 7. Likewise, step 3 can be incorporated into step 10 then reduced to $O(i * n)$. Thus, the total time required to solve the problem in parallel is

$$T_p = \overbrace{\Theta\left(\frac{i * n^2}{p} + i\right)}^{computation} + \overbrace{\Theta\left(i * n\right)}^{communication}$$

Thus,

$$T_o = \Theta\left(i * n\right)$$

$$W = \Theta\left(\frac{i * n^2}{p} + i\right)$$

where T_o and W are the overhead and useful work performed respectively.

It is apparent that the time required to exchange of subdomain matrices dominates the communications overhead, T_o. Since the serial algorithm is $\Theta(i*n^2)$, the processor/time product is optimal for $n > p$.

PROGRAM Parallel Beam-Warming
PROGRAM INPUTS: initial values for flow solution to include number of grid points, Mach number, and convergence criteria.

1. *Partition the computational grid.* $O(\frac{n}{p})$
2. *Initialize the flow variables.* Initialize all flow variables, matrices, and grid to initial conditions. Initialization of the n elements in the matrices and grids require $O(\frac{n}{p})$ time.
3. *Send initialization data to all processors.* Send the data read from the input file to all processors. This takes $O(\sqrt{n})$ time.
4. *Compute metrics of transformation.* Transform the physical grid into the computational grid. Mapping the two-dimensional grid requires $O(\frac{n}{p})$ time.
5. *Compute local time step.* Calculate the maximum local time step allowable for each grid point. Sweeping through the two-dimensional grid requires $O(\frac{n}{p})$ time.
6. *Loop* Continue until maximum residual is less than required convergence criteria.
7. *Sweep in η direction.* Solve (19) in η plane, computing right-hand side (RHS) terms for η direction. This requires $O(\frac{i*n}{p})$ time.
8. *Sweep in ξ direction.* Sweep through ξ plane and solve (19) in ξ plane, computing right-hand side (RHS) terms for ξ direction. This requires $O(\frac{i*n}{p})$ time.
9. *Global load of A and b matrices into SES solver.* Use SES to assemble the A and b matrices. This global exchange of values requires $O(i * n)$ time.
10. *Solve $Ax = b$ using SES.* Use SES to solve system of equations. This requires $O(\frac{i*n^2}{p})$ time.
11. *Global distribution of x.* x is distributed on the processors using all-to-all personalized communication. The time required for this operation is $O(i * \sqrt{n})$.
12. *Implement boundary conditions.* Implements the boundary conditions for the geometry of interest. This requires $O(1)$ time per iteration or $O(i)$ for the entire solution.
13. *End time increment.*

Figure 23. Time Complexity of Parallel Beam-Warming Algorithm

The predicted speedup is for this implementation of the Beam-Warming algorithm is

$$S = \frac{\Theta(i * n^2)}{\Theta(\frac{i*n^2}{p}) + \Theta(i * n)}$$

The projected efficiency for the Beam-Warming algorithm is

$$E_{bw} = \frac{\frac{\Theta(i*n^2)}{\Theta(\frac{i*n^2}{p})+\Theta(i*p^{\frac{3}{2}})}}{p} = i\left(\frac{n^2 p}{n^2+n}\right)$$

This results in the isoefficiency function for this implementation of the Beam-Warming algorithm is

$$n = p$$

Thus, as the size of the problem is increased by n, the number of processors required to provide the same efficiency is p.

Figures 24 and 25 show the parallel flow of the Beam-Warming code.

4.3 Algorithm Test Design

The following measurements are required to properly understand an algorithm's performance:

- **Fixed-workload speedup** - The number of processors are increased while keeping the amount of work constant. This measures the maximum parallelism within a single parallel program.

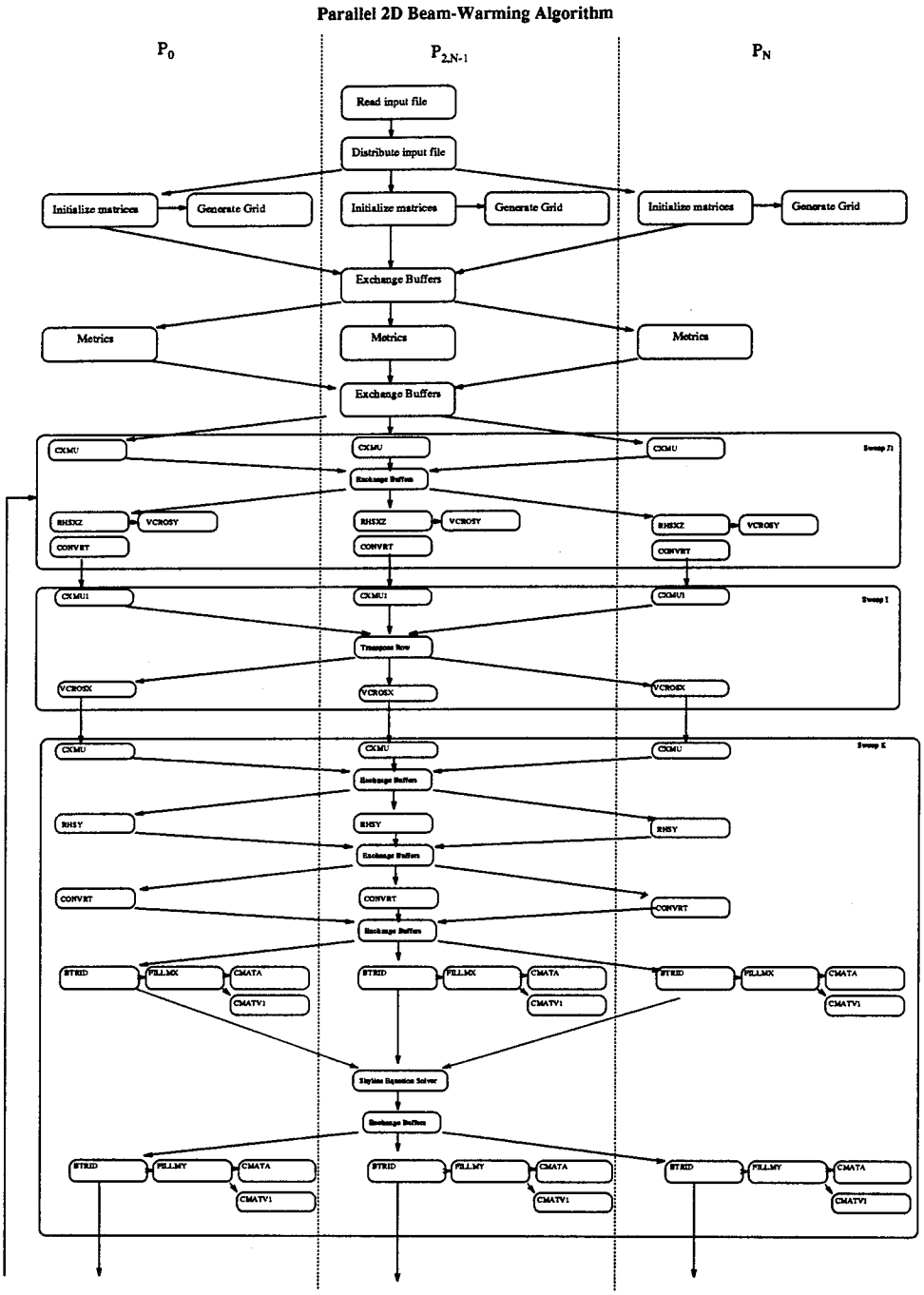


Figure 24. Parallel Flow of Beam-Warming Code.

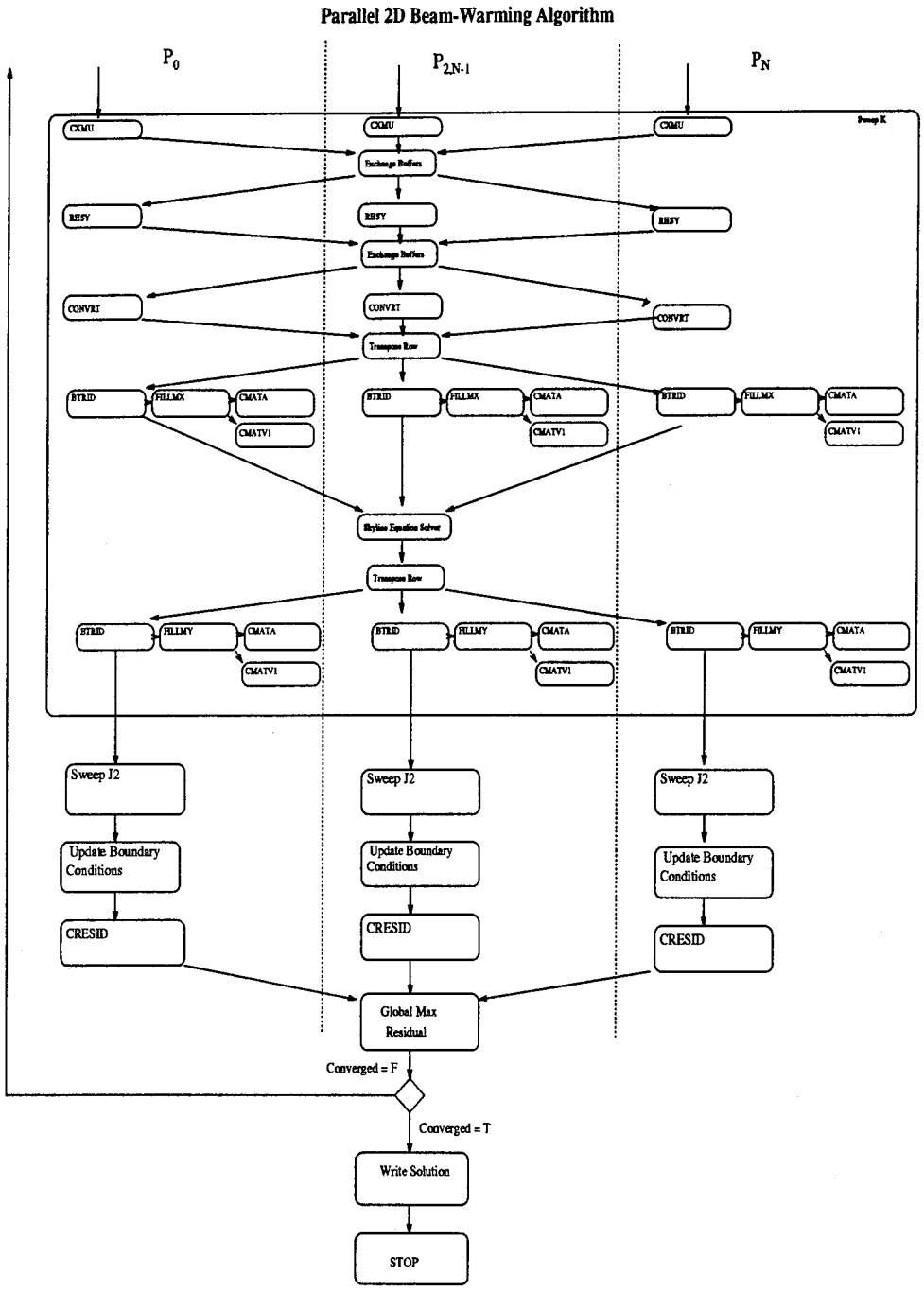


Figure 25. Parallel Flow of Beam-Warming Code (cont'd).

- **Memory-bound speedup** - The problem size is scaled with the number of processors and the relationship measured. Memory-bounded speedup is defined as

$$S_{mb} = n\frac{T(1)}{T(n)}$$

This measures the scalability of the program.

- **Fixed-time speedup** - The number of jobs was increased as processors were added while keeping execution time constant. This shows the relationship between scaling a system and problem size given a fixed solution time. Figure 26 (22) shows the three types of speedup.

- **Efficiency** - This is an indication of the actual degree of speedup performance as compared with the ideal. A comparison of the expected and actual efficiencies was made.

- **Execution time versus Accuracy** - A comparison between relaxation of convergence criteria and solution generation time was made.

The test matrices shown in Figures 27,28, and 29 are developed in order to measure the algorithm performance.

4.4 Summary

A design process and implementation details are presented. A UNITY description describing the inherent parallelism within each algorithm and the actual parallel implementations detail the methods used to extract the maximum parallelism from the given

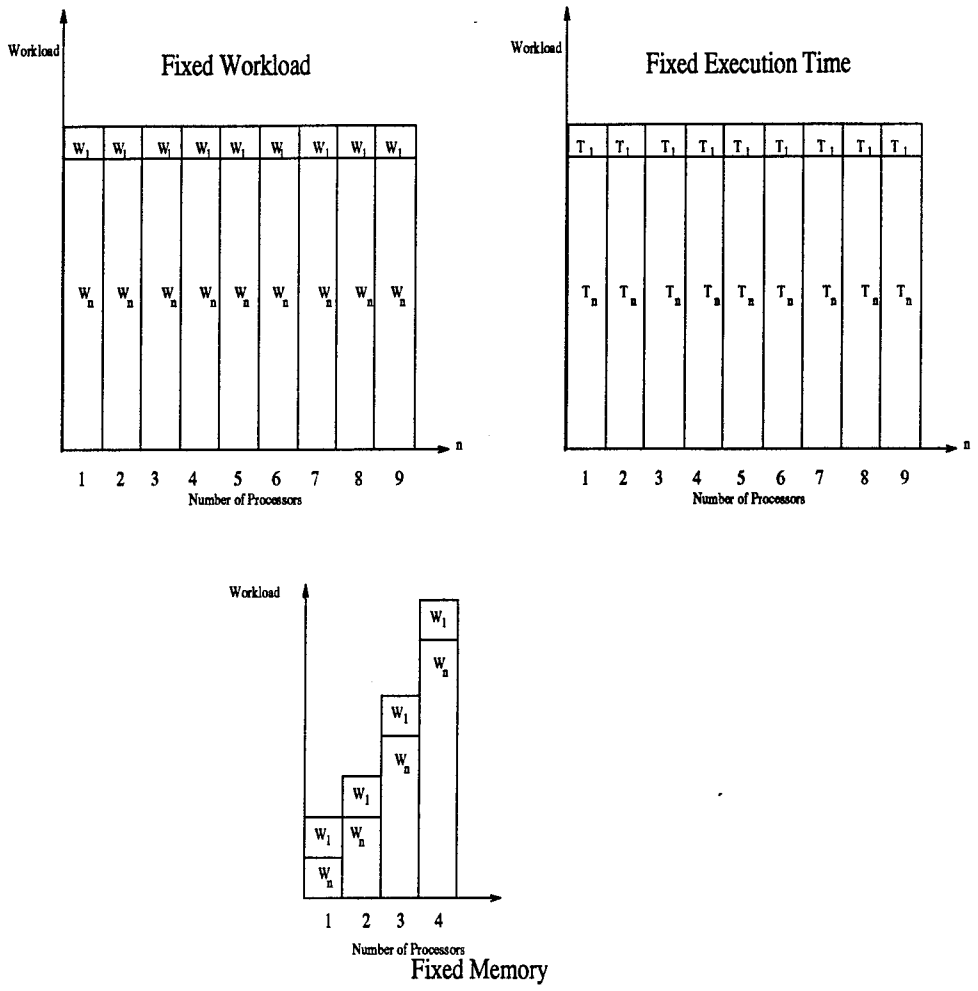


Figure 26. Memory, Time, and Workload Bound Speedup

Fixed Memory Test Matrix

Grid Size	Processors	Accuracy	
		1E-3	1E-4
21x21	1,2,4,8	✓	✓
32x32	1,2,4,8,16	✓	✓
42x42	1,2,4,8,16	✓	✓
81x81	1,2,4,8,16,32	✓	✓
64x64	1,2,4,8,16,32	✓	✓
128x128	1,2,4,8,16,32,64	✓	✓

Figure 27. Fixed-workload Test Matrix

Memory-bound Test Matrix

Grid Size	Processors	Accuracy 1E-3	Accuracy 1E-4
21x21	1	✓	✓
32x32	2	✓	✓
42x42	4	✓	✓
81x81	8	✓	✓
64x64	16	✓	✓
128x128	32	✓	✓

Figure 28. Memory-bound Test Matrix

Fixed-time Test Matrix

Grid Size	Processors	Accuracy (1E-4)
21x21	1	✓
27x27	2	✓
34x34	4	✓
42x42	8	✓
53x53	16	✓
67x67	32	✓

Fixed-time Test Matrix

Grid Size	Processors	Accuracy (1E-3)
21x21	1	✓
27x27	2	✓
34x34	4	✓
42x42	8	✓
57x57	16	✓
70x70	32	✓

Figure 29. Fixed-time Test Matrix

algorithms. Finally, a test design is presented that enables the proper measurement of the parallel implementations effectiveness.

V. Results

Results for the parallel Roe scheme and the parallel Beam-Warming implementations are provided. Results are validated and compared to expected values derived from calculations presented in Chapter 4.

5.1 Roe Scheme Performance Analysis

The following sections evaluate the parallel performance of the Roe scheme as implemented on the Intel Paragon. Compiler directives, code validation, and a comparison of actual versus expected results are discussed in detail. All tests are measured in seconds returned from the *dclock* system call.

5.2 Compiler Optimizations

Various compiler directives were tested on the serial and parallel codes for a problem size of 1024 grid points (32x32 grid) in order to determine the settings that would produce the best performing code. This test ensured that the best serial algorithm was used as a benchmark for determining speedup. There was an obvious improvement in code performance from optimization level 2 (-O2) to optimization level 3 (-O3) and a smaller increase between optimization level 3 and level 4. Inclusion of the vectorization switch (-Mvect) appeared to have little impact on code performance. The most aggressive compilation options,-O4 and -Mvect, were chosen because of their potential ability to increase code performance for larger problem sizes. The results are shown in Figure 30.

Complier Option / Processors	-Mvect -O4 (sec)	-O4 (sec)	-Mvect -O3 (sec)	-O3 (sec)	-Mvect -O2 (sec)	-O2 (sec)
1	4545	4537	4549	4551	4692	4685
2	2257	2264	2261	2254	2340	2337
4	1206	1206	1213	1206	1260	1254
8	613	610	615	610	634	641
16	316	312	315	313	325	321

Compiler Options:

-O2 Some register allocation is performed.Traditional scalar optimizations perfromed by global optimizer.
-O3 All level 2 optimizations in addition to software pipelineing.
-O4 All level 3 optimizations with more aggressive register allocation for software pipelined loops.
-Mvect Vectorization performed

Figure 30. Effects of Compiler Flags on Roe Scheme Performance.

5.3 Code Validation

The accepted solution for the shock wave standoff distance was calculated using the equation

$$\frac{\delta}{R} = .386exp\left[\frac{4.67}{M_\infty^2}\right] \tag{26}$$

from (2) where $\frac{\delta}{R}$ is the standoff distance for the shockwave. The accepted solution was 3.08 diameters from the cylinder surface. Figure 31 shows the shock wave appearing at 1.14 diameters from the cylinder sufrace. A comparison between the numerical solution and the accepted solution shows the serial solution cannot be considered validated since the error was not within the expected accuracy of a first-order accurate algorithm. The error was most likely due a combination of grid constriction and grid fineness.

Validation of the parallel Roe scheme solution was accomplished by comparing the number of iterations to converge, maximum residual at each iteration, and numerical data

to those from the serial algorithm. The validation runs were performed using various grid sizes and two different convergence criteria. Both codes were expected to produce the same result.

Comparison of the number of iterations to converge and maximum residuals for the serial and parallel codes proved to be identical out to over sixteen decimal places. Figures 33 and 34 show the convergence histories flow over a cylinder using a 64x64 grid with a convergence tolerance of 1×10^{-3}. Because they were identical only one is shown. Figures 32 and 31 show the velocity and pressure output for flow over a cylinder (64x64 grid). Since the solutions for each of the above were identical to the serial output the algorithm was assumed validated and no additional solutions for different grid sizes presented.

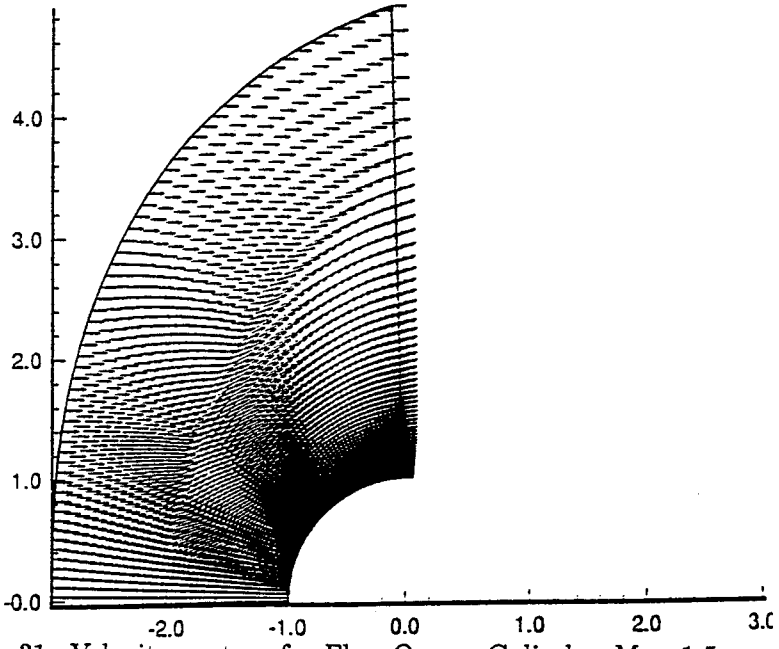


Figure 31. Velocity vectors for Flow Over a Cylinder, M = 1.5 .

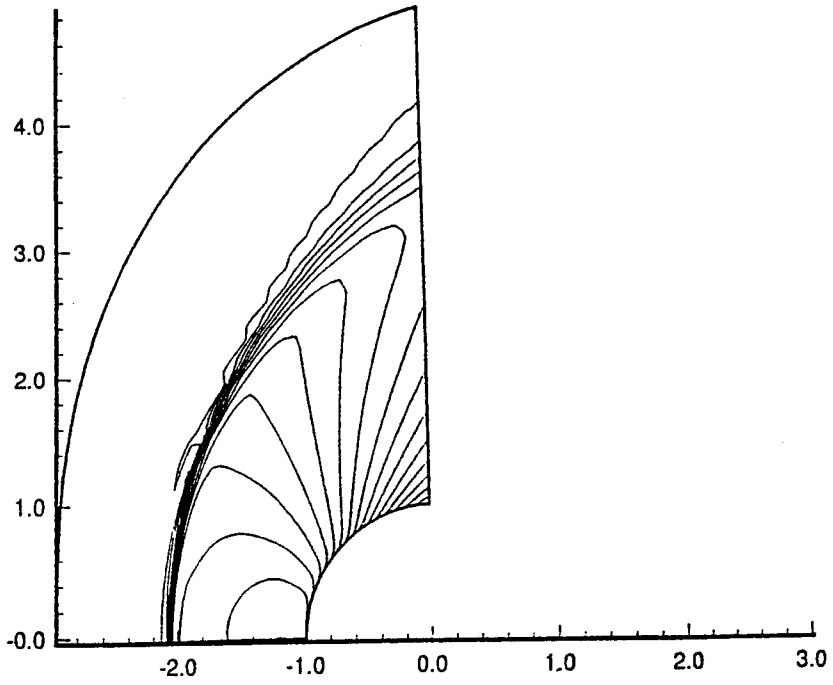


Figure 32. Pressure Contours for Flow Over a Cylinder, M = 1.5.

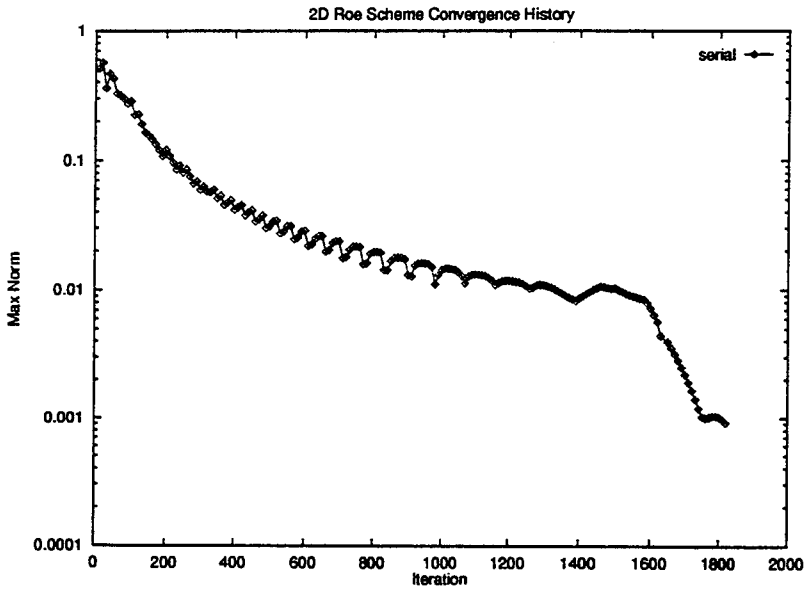


Figure 33. Convergence History for Serial Roe Scheme.

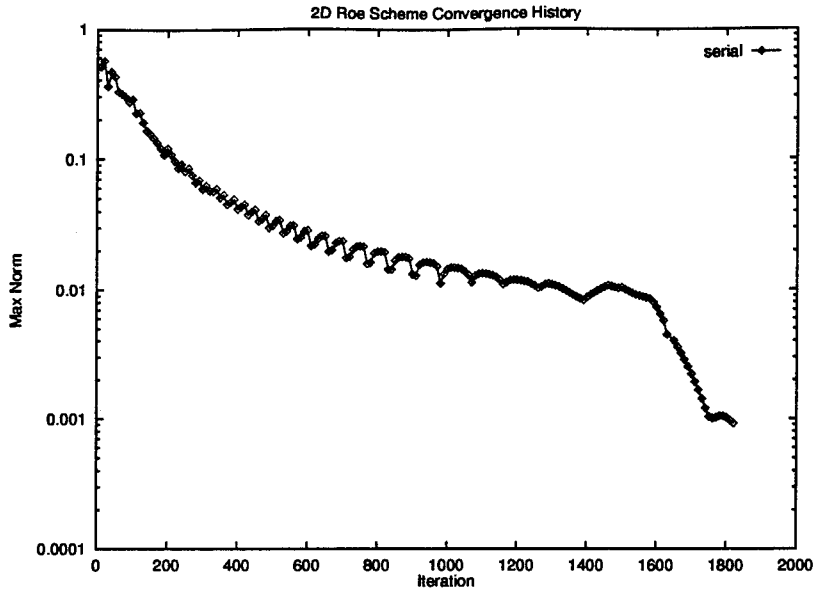


Figure 34. Convergence History for Parallel Roe Scheme.

5.4 Parallel Performance

Figures 35 and 36 contain the iterations until convergence and CPU times for convergence tolerances of 1×10^{-4} and 1×10^{-3} respectively. Figures 37 shows the speedup of the fixed-workload algorithm. This data corresponds to the fixed-workload tests specified in Figure 27.

Figure 37 shows the fixed-workload speedup. The results show excellent speedup for up to four processors then slowly diminishing performance out to the maximum number of processors applied to each problem size. This result was understandable since the addition of processors while keeping the problem size fixed reduced the computation to communication ratio. An anomaly occurred in the speedup measurement for the 128×128 case. While the speedup was initially less for smaller problem sizes, it performed nearly linearly between $p = 2$ and $p = 32$. It appears that this was because the problem size

Fixed-Workload Test Results
(1E-4)

Grid Size	Processors	Iterations to Converge	Time to Converge(sec)
21x21	1	2610	1428
	2		762
	4		385
	8		235
32x32	1	3260	4534
	2		2257
	4		1206
	8		610
	16		316
42x42	1	3730	8992
	2		4590
	4		2533
	8		1398
	16		702
64x64	1	4740	29506
	2		14511
	4		7460
	8		3765
	16		1900
	32		1024
81x81	1	Did	
	2	Not	
	4	Converge	
	8		
	16		
	32		
128x128	1	5620	129016
	2		71676
	4		36444
	8		18262
	16		9654
	32		4598
	64		2326

Figure 35. Fixed-Workload Performance of Roe Scheme (1E-4).

Fixed-Workload Test Results
(1E-3)

Grid Size	Processors	Iterations to Converge	Time to Converge(sec)
21x21	1	1100	604
	2		324
	4		166
	8		101
32x32	1	1410	1977
	2		984
	4		528
	8		267
	16		137
42x42	1	1580	3803
	2		1955
	4		1086
	8		597
	16		305
64x64	1	1820	11684
	2		5592
	4		2898
	8		1450
	16		734
	32		373
81x81	1	2130	22622
	2		11809
	4		5909
	8		3101
	16		1698
	32		856
128x128	1	3350	72144
	2		43588
	4		21748
	8		10896
	16		5463
	32		2746
	64		1465

Figure 36. Fixed-Workload Performance of Roe Scheme (1E-3).

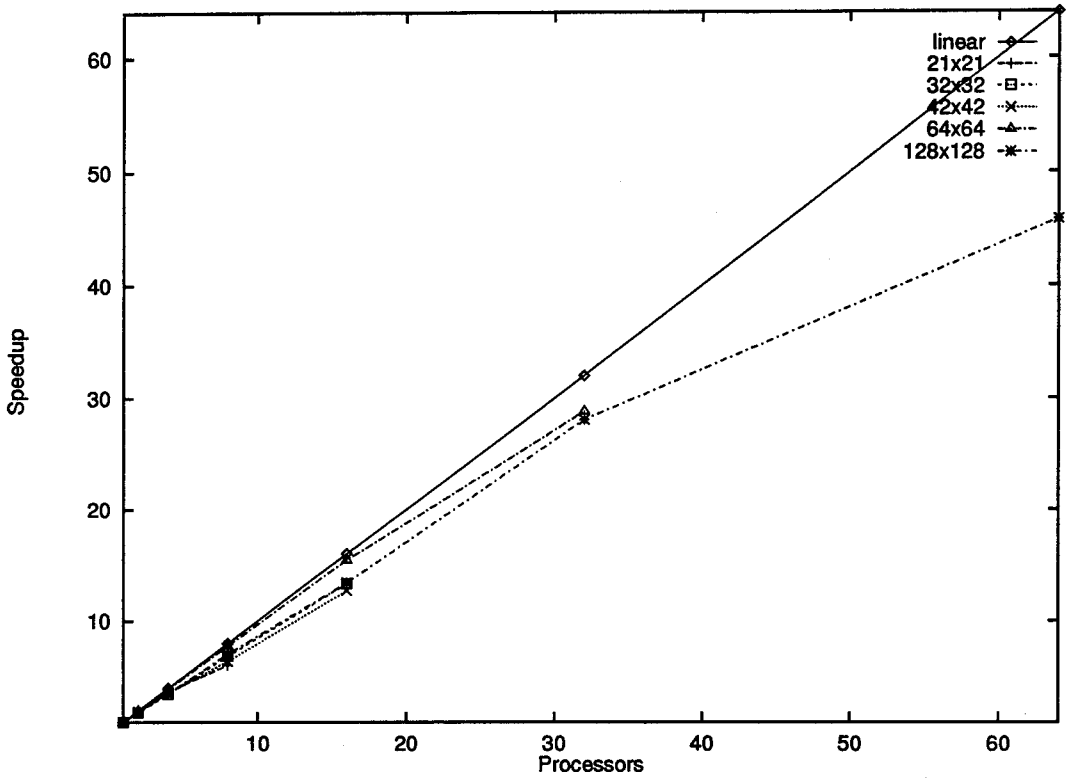


Figure 37. Fixed-Workload Speedup of Roe Scheme.

became big enough for the vectorization and software pipelining to improve the serial run-time, thus reducing the speedup.

Figure 38 contains the results of the memory-bound test matrix specified in Figure 28. The result shows that the problem size does scale better than linearly with the number of processors. This is expected since the isoefficiency calculation stipulates that an increase of n in the problem size requires p^2 processors to maintain the same efficiency.

Memory-Bound Test Results

Grid Size	Processors	Iterations to Converge		Time to Converge(sec)	
		(1E-4)	(1E-3)	(1E-4)	(1E-3)
21x21	1	2610	1100	1428	604
32x32	2	3260	1410	2257	1977
42x42	4	3730	1580	2533	3803
64x64	8	4740	1820	3765	11684
81x81	16	N/A	2130	N/A	22622
128x128	32	5620	3350	4598	72144

Figure 38. Memory-Bound Performance of Roe Scheme.

Similarly, the isoefficiency calculation can predict the performance for fixed-time speedup. The results of the fixed-time test matrix specified in Figure 29 are shown in Figure 40. Figure 41 shows the predicted efficiency as compared to the actual efficiency and confirms the isoefficiency calculation.

It is apparent that a decrease in the computation to communication ratio results in decreasing efficiency. Thus, the efficiency of the Roe scheme should follow a generally

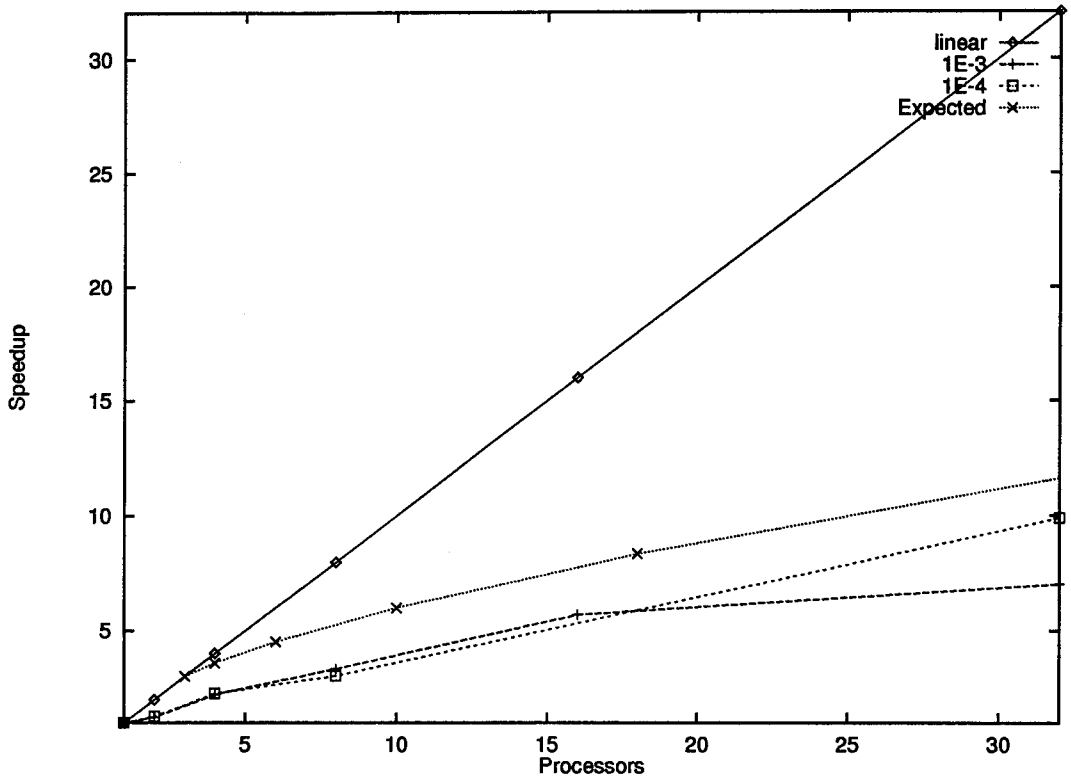


Figure 39. Memory-Bound Speedup of Roe Scheme.

Fixed-Time Test Matrix

(1E-4)

Grid Size	Processors	Iterations	Time to Converge (sec)
21x21	1	2610	1428
27x27	2	2900	1486
34x34	4	3420	1506
42x42	8	3730	1398
53x53	16	4340	1381
67x67	32	4790	1467

Fixed-Time Test Matrix

(1E-3)

Grid Size	Processors	Iterations	Time to Converge (sec)
21x21	1	1110	604
27x27	2	1290	644
34x34	4	1350	649
42x42	8	1410	597
57x57	16	1750	609
70x70	32	1870	607

Figure 40. Fixed-Time Performance of Roe Scheme.

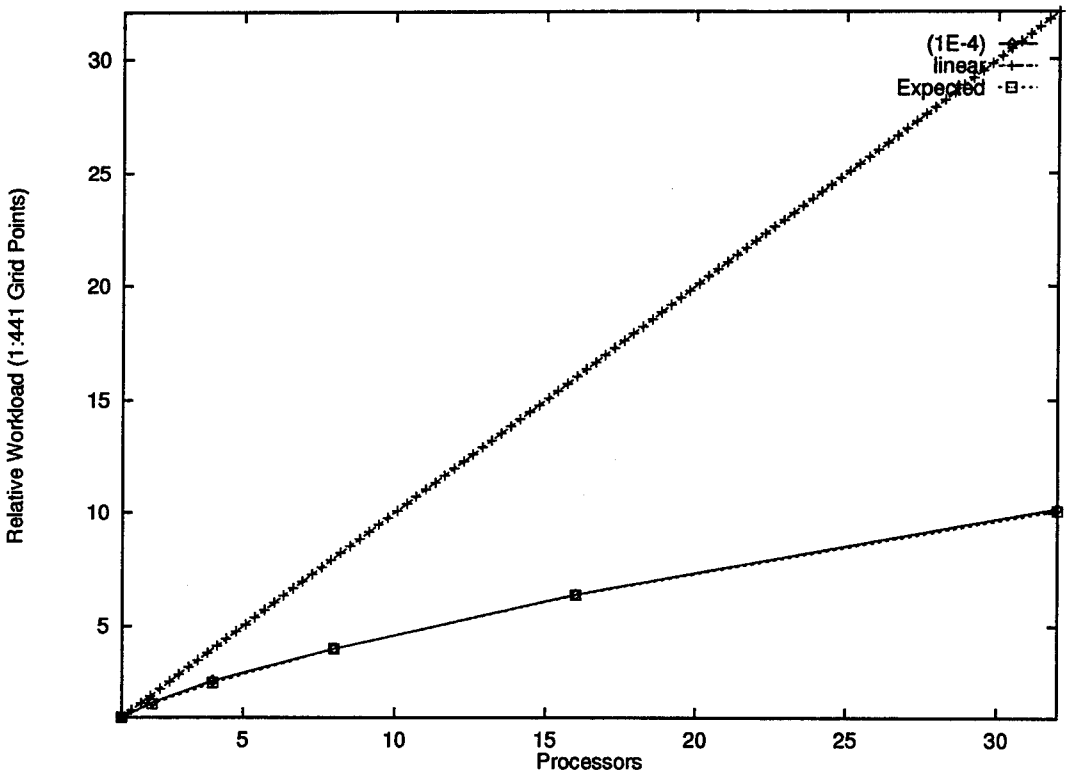


Figure 41. Fixed-Time Speedup of Roe Scheme.

decreasing pattern as the number of processors are increased while keeping the problem size constant. Figure 42 shows the actual measured efficiencies.

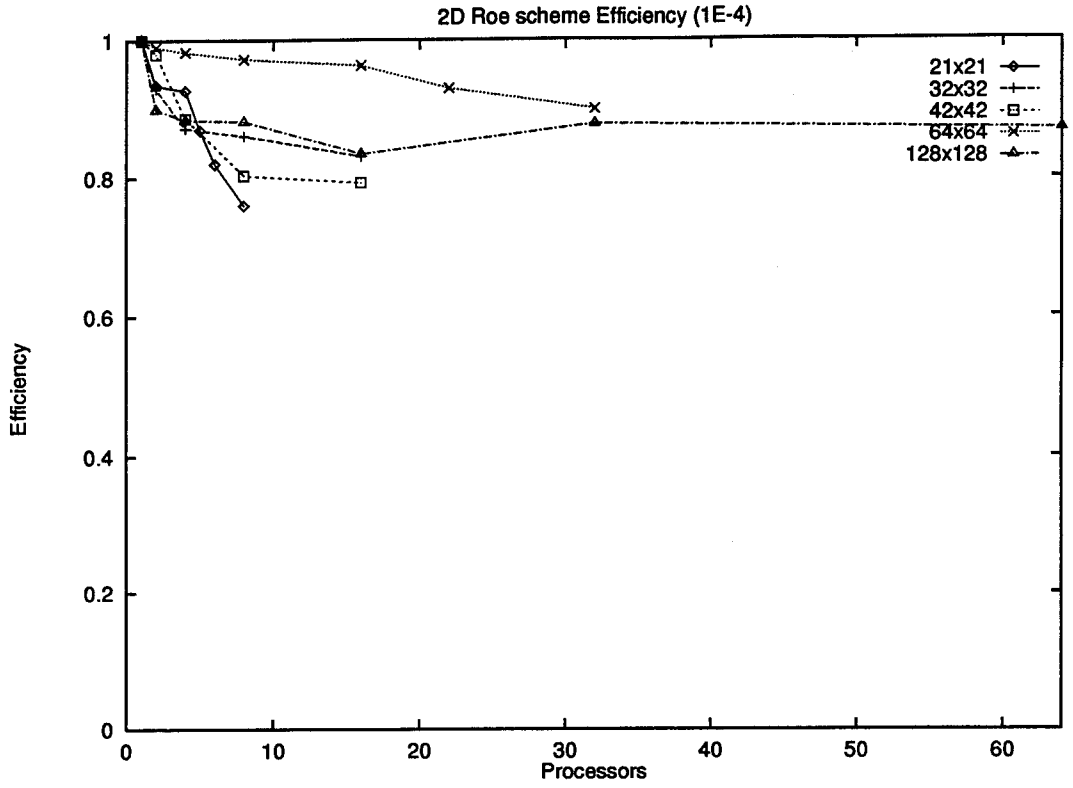


Figure 42. Efficiency of Roe Scheme.

5.5 Execution Time versus Convergence Tolerance

Comparison of the execution times for the two levels of convergence criteria measured shows the tradeoff between the two for the Roe scheme. In general, convergence criteria of 1×10^{-3} required between 38% (64×64) and 59% (128×128) of the iterations required for a convergence criteria of 1×10^{-4}. As a result, the actual time to converge differs by the same amount. Figure 43 shows the comparison in run times of various problem sizes for the two levels of tolerance.

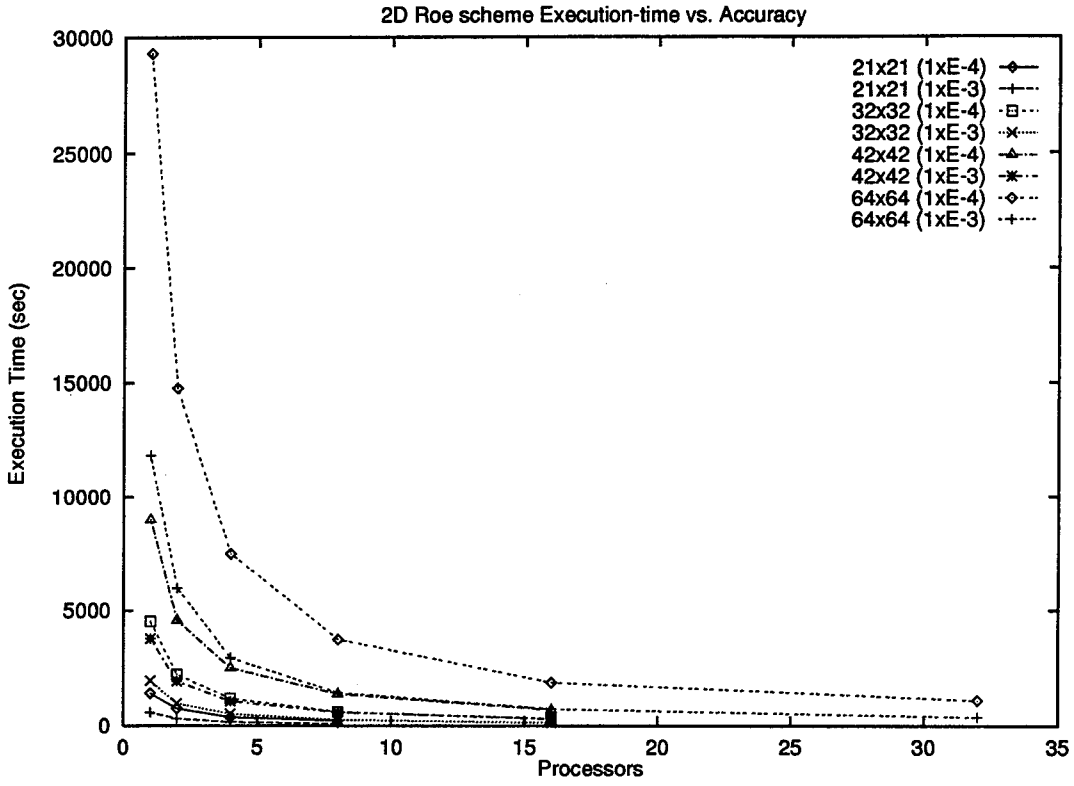


Figure 43. Execution Time versus Accuracy.

5.6 Beam-Warming Algorithm Performance Analysis

The parallel algorithms required to implement the Beam-Warming algorithm were completed and tested. Parallelization of the Beam-Warming algorithm was not completed due to a failure in arriving at a correct numerical solution for the serial algorithm using SES. Using existing data structures for solving the linear system of equations were possibly the cause of the error. Due to time constraints, the error could not be isolated. The parallel analysis from Chapter 4 indicates that based on the isoefficiency, the parallelized Beam-Warming algorithm should be very efficient and show excellent scalability on the Paragon.

VI. Conclusion and Recommendations

6.1 Conclusion

The objectives of this investigation were

- Analyze and implement techniques required to parallelize CFD problems.
- Estimate the general behavior of performance based on the complexity analysis.
- Measure the performance of the parallel explicit algorithm for supersonic flow over a cylinder.
- Compare actual results to the estimates for validation.

All objectives were met. A one-dimensional partitioning algorithm, index adjustment, and a buffer swapping algorithm were constructed,tested, and validated. These routines enabled the serial Roe Scheme to run in parallel with very few code modifications.

A complete analysis of the order of complexity for the Roe scheme and Beam-Warming algorithm was developed and the characteristic performance estimated. The performance of the algorithm could be traced directly to the isoefficiency function which describes the inherent scalability of an algorithm. Given the isoefficiency function, the general performance for fixed-memory, fixed-time, and memory-bounded speedup were determined.

Solution time was measured for two levels of convergence tolerance for inviscid, supersonic flow over a cylinder. These measurements for fixed-workload speedup, fixed-memory speedup, and fixed-time speedup were compared to the estimated performance of the algorithm in order to determine if the algorithms were characterized correctly.

While the tools generated are important in easing the parallelization process, the most important finding in this research was that by rigorously characterizing the implementation of a parallel code, the general performance of the algorithm can be predicted. Typically, parallel performance is characterized by only measuring speedup which results in only one perspective of an algorithm's potential performance. By including the isoefficiency measurement, an algorithm's performance in differing environments (fixed-memory, fixed-time, memory-bound) can be correctly characterized.

6.2 Future Research

The following are areas that should be explored for improving parallel CFD performance:

1. Implementation of the parallel Roe Scheme on a hypercube architecture and comparison to the predicted performance.
2. Parallelization of the Beam-Warming algorithm and comparison against the estimated performance.
3. Parallelization of the Thomas algorithm should be further explored in order to reduce the "order of" complexity for solving a block tridiagonal system.
4. Study of an uneven partitioning algorithm that takes into account the uneven number of operations that must be performed on each grid point would increase the efficiency of an algorithm. For example, in a 1D partitioned domain, the outer-most processors must implement additional boundary conditions. This forces the other processors to wait wait at some synchronization point because they perform fewer operations.

5. Strategies for partitioning and solving the linear system of equations that must be implemented for geometries that do not require a full sweep of the domain (i.e. up a step, over a step) should be studied.

Appendix A: Parallel Subroutines

The following are code listings for the subroutines generated for parallelization of the 2D Roe Scheme.

Domain Decomposition

PARTITION1D decomposes the domain in one dimension. Two calls to *PARTITION1D* are required in order to calculate the starting and ending indices of each processor's local partition. The following subroutine calls will partition the domain in one dimension:

call partition1d(1,IDM,iam,nodes,i1)

call partition1d(1, IDM,nextnode, nodes,i2)

where IDM is the max length of the dimension, *iam* is the node number, *nodes* is the total number of nodes to partition across, $i1$ is the starting index of the local partition, and $i2$ is the starting index of the next partition. Decrementing $i2$ by one gives the ending index of the local partition. Note that the decrement is handled by the *INDEXADJUST* subroutine. A second series of calls to *PARTITION1D* with the maximum length of another dimension will result in a partitioning in a second dimension.

```
c v2.0 SUBROUTINE PARTITION1D
c Partition one dimension of an array into a 1-D array. Compute the index
c range (i1-i2) of a part of a 1-D array (two calls required). The first
c call calculates the starting index of each partition, the second call
c calculates the end point of each partition. The indices will
c fall between M and N, the global limits of the indices.
c
c       M <= i1 <= i2 <= N
c
        subroutine partition1d(start, end, iam, partitions,i)
c
        integer start, end, partitions,total, size, rem, extra,i
c
        total = end - start + 1
        size = total / partitions
```

```
      rem = mod(total, partitions)
      extra = min(rem, iam)
c
      i = start + iam*size + extra
      return
      end
```

Index Adjustment

ADJUSTINDEX adjusts the starting and ending partition indices in order to allow the serial code to run without significant modification. The $ISP1$,$IEM1$,$JSP1$,and $JEM1$ indices are adjusted based on whether a processor has an "end" or "middle" partition as explained in Chapter 4. Additionally, each processor is assigned a "blocktype" which is used to determine what metrics and boundary condtions apply to which partitions. The following call to *ADJUSTINDEX* will automatically adjust the indices:

```
call indexadjust(i1,i2,is,isp1,iem1,ie,js,jsp1,jem1,je,blocktype)
```

where $i1$ and $i2$ are the indices returned from *PARTITION1D*. The subroutine returns partition indices IS,$ISP1$,$IEM1$,IE,JS,$JSP1$,$JEM1$, and JE as well as each partition's "blocktype." The code listing for *ADJUSTINDEX* is as follows:

```
c v2.0 SUBROUTINE INDEXADJUST
c Adjust indices and determine blocktype
c
      subroutine indexadjust(i1,i2,is,isp1,iem1,ie,js,jsp1,jem1,je,
     & blocktype)
c
      integer i1,i2, is,isp1,iem1,ie,js,jsp1,jem1,je,
     & blocktype
c
c Calculate indices
      i2 = i2-1
      is = i1
      ie = i2
```

```
        js = 1
        je = jdm
        iem1 = ie-1
        isp1 = is+1
        jsp1 = js+1
        jem1 = je-1
c
c Determine blocktype and adjust indices based on which blocktype is on
c each processor.
        if(is.eq.1) then
            iem1 = ie
            blocktype = -1
        else if (ie.eq.idm) then
            isp1 = is
            blocktype = 1
            else
                isp1 = is
                iem1 = ie
                blocktype = 0
            end if
        return
        end
```

Buffer Exchange

EXCHANGEBUFFS automates the buffer exchange with adjacent partitions. By specifying a "mode" and the data structure on which to operate, each partition automatically copies the data into a message, exchanges to data with neighboring partitions, and places the data into the correct buffers. Examples of 2-D and 3-D buffer exchange are as follows:

```
      call exchangebuffs(mode,is,ie,js,je,iam,etax,xnum,ynum,blocktype,buffeasts,
     & buffwests,buffeastr,buffwestr,prevnode,nextnode)

      call exchangebuffs(mode,is,ie,js,je,iam,capu,xnum,ynum,blocktype,buffeasts,
     & buffwests,buffeastr,buffwestr,prevnode,nextnode)
```

where *mode* determines whether the exchange operates on a 2-D (*mode* = 0) or a 3-D

($mode > 0$) data structure, *etax* and *capu* are examples of the data structure on which the operation takes place,*xnum* and *ynum* are the maximum length of each dimension,*blocktype* is the "type" of partition on a processor,$buffeasts$,$buffwests$,$buffeastr$, and $buffwestr$ are send and receive buffers, *prevnode* and *nextnode* are the previous and next processor number, and IS,IE,JS, and JE are dimension indices.

```
      subroutine exchangebuffs(mode,is,ie,js,je,iam,xarray,xv,yv,blocktype,
     & buffeasts,buffwests,buffeastr,buffwestr,
     & prevnode,nextnode)
c
c v1.0 -   Exchange edge data with neighboring processor
c          This is done only in I dimension due to 1D division
c      -   The size of the block exchanged is the J dimension *8
c          Change jdm to (N/nodes)*8 if partitioning in J dimension
c          as well.
c      -   Separate buffers are used for send and receive buffers mainly
c          for ease of debugging. If memory is a problem, send and receive
c          buffers can be collapsed into one.
c
c v1.2 -   message types are assigned as follows:
c          from right (fr) : 1000
c          from left (fl) : 2000
c          to left (tl) : 1000
c          to right (tr) : 2000
c
c          message pairs are opposites to properly match between
c          nodes that are sending and nodes that are receiving.(ie tr/fr= 1000)
c          Basing the message number on node number plus some unique message
c          type will ensure unique message types.
c v1.3 -   Mode tells whether a 2D or 3D buffer swap is to take place. This
c          allows all buffer swaps to use one routine.
c      -   Mode = 0 is for 2D buffer swap,anything else is 3D buffer swap
c
      integer mode,is,ie,js,je,x,iam,blocktype,
     & iep1,N,xv,yv,fr,fl,tr,tl,prevnode,nextnode
      real xarray(0:xv+1,yv),buffeasts(jdm),buffwests(jdm),
     & buffeastr(jdm), buffwestr(jdm)
c
      N=je
      iep1=ie+1
      ism1 = is -1
      fr = 1000
```

```
      fl = 2000
      tl = 1000
      tr = 2000
c
      if(blocktype.eq.-1) then
c v1.0 -  This is the leftmost block, only one send and one receive.
c     -  Form buffer arrays to send.
         if (mode.eq.0) then
            Do 4 J = js,je
               buffeasts(j) = xarray(ie,j)
4           Continue
         else
            Do 41 I = 1,4
               Do 41 J = js,je
                  buffereasts(j,i) = xarray(ie,j,i)
41          Continue
         end if
c send matrix to block on right
         call csend(nextnode+tr,buffeasts(js),N*8,nextnode,0)
c receive matrix from block on right
         call crecv(iam+fr,buffeastr(js),N*8)
c copy message into buffer
         if (mode.eq.0) then
            Do 42 J = js,je
               xarray(iep1,j) = buffeastr(j)
42          Continue
         else
            Do 43 I = 1,4
               Do 43 J = js,je
                  bufferwests(j,i) = xarray(is,j,i)
43          Continue
         end if
      else if(blocktype.eq.1) then
c v1.0 -  This is the rightmost block, only one send and one receive.
c     -  Form buffer arrays to send.
         if (mode.eq.0) then
            Do 51 J = js,je
               buffwests(j) = xarray(is,j)
51          Continue
         else
            Do 52 I = 1,4
               Do 52 J = js,je
                  buffereasts(j,i) = xarray(ie,j,i)
52          Continue
         end if
```

```
c send matrix to block on left
            call csend(prevnode+tl,buffwests(js),N*8,prevnode,0)
c receive matrix from block on left
            call crecv(iam+fl,buffwestr(js),N*8)
            if (mode.eq.0) then
                Do 53 J = js,je
                    xarray(ism1,j)= buffwestr(j)
        53          Continue
            else
                Do 54 I = 1,4
                    Do 54 J = js,je
                        xarray(ism1,j,i) = bufferwestr(j,i)
        54          Continue
            end if
            else
c v1.0 -    This is the regular block with two sends and receives to
c           do.
c       -   Form buffer arrays to send.
                if (mode.eq.0) then
                    Do 6 J = js,je
                        buffeasts(j) = xarray(ie,j)
                        buffwests(j) = xarray(is,j)
        6           Continue
                else
                    Do 61 I = 1,4
                        Do 61 J = js,je
                            buffereasts(j,i) = xarray(ie,j,i)
                            bufferwests(j,i) = xarray(is,j,i)
        61          Continue
                end if
c send matrix to block on left
                call csend(prevnode+tl,buffwests(js),N*8,prevnode,0)
c send matrix to block on right
                call csend(nextnode+tr,buffeasts(js),N*8,nextnode,0)
c receive matrix from block on left
                call crecv(iam+fl,buffwestr(js),N*8)
c receive matrix from block on right
                call crecv(iam+fr,buffeastr(js),N*8)
                if (mode.eq.0) then
                    Do 62 J = js,je
                        xarray(iep1,j)= buffeastr(j)
                        xarray(ism1,j) = buffwestr(j)
        62          Continue
                else
                    Do 63 I = 1,4
```

```
                Do 63 J = js,je
                    xarray(ism1,j,i) = bufferwestr(j,i)
                    xarray(iep1,j,i) = buffereastr(j,i)
63          Continue
        end if
        end if
return
end
```

Bibliography

1. *Paragon Application Tools User's Guide.*
2. Anderson, D. A., et al. "Computational Fluid Mechanics and Heat Transfer," (1984).
3. Angelaccio, M. and M. Colajanni. "Subcube matrix decomposition: A unifying view for LU factorization on multicomputers," *Parallel Computing*, (20):257–270 (1994).
4. Argawal, R. "CFD Short Course." 1993.
5. Barth, T.J. "Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations," (May 1992).
6. Beran, P and T. Buter. "AERO 752 lecture notes." 1992.
7. Bramble, J., et al. "The construction of preconditioners for elliptic problems by substructuring I," *Math. Comp.*, (47):104–134 (1986).
8. Chan, T. *Domain Decomposition Algorithms and Computational Fluid Dynamics.* Technical Report 88.21, RIACS Technical Report, 1988.
9. Chandy, K and J. Misra. *Parallel Program Design.* Addison-Wesley Publishing Company, Inc., 1988.
10. Churchitser, E., et al. "Solution of the Euler and Navier-Stokes Equations on MIMD Distributed Memory Multiprocessors Using Cyclic Reduction." Number AIAA 92-0561 in 30th Aerospace Sciences Meeting and Exhibit. January 1992.
11. Chyczewski, T., et al. *Solution of the Euler and Navier-Stokes Equations on Parallel Processor using a Transposed/Thomas ADI Algorithm.* Technical Report AIAA 93-3310-CP, 1993.
12. Daniel, D.C. and J.R.Graham. ""Integration of Supercomputers with Aerospace Ground Test Facilities.". January 1992.
13. Davis, D. *A Parallel Computational Fluid Dynamics Unstructured Grid Generator.* MS thesis, Air Force Institute of Technology, 1993.
14. Dryja, M. and O.B. Widlund. *An additive variant of the Schwarz alternating method for the case of many subregions.* Technical Report, Department of Computer Science, Courant Institute, December 1987.
15. Farhat, C. and F. Roux. "An Unconventional Domain Decomposition Method for an Efficient Parallel Solution of Large-scale Finite Element Systems," *J. Sci. Stat. Comput.*, *13*(1):379–396 (January 1992).
16. Gropp, W. and D. Keyes. "Domain Decompostion on Parallel Computers." *Proceedings of the Second International Symposium on Domain Decomposition Methods.* 260–268. January 1988.
17. Gropp, W. and E. Smith. "Computational Fluid Dynamics on Parallel Processors," *Computers & Fluids*, *18*(3):289–304 (1990).
18. Hammond, S. *Mapping Unstructured Grid Computations on Massively Parallel Computers.* PhD dissertation, Rensselaer Polytechnic Institute, 1992.

19. Henderson., B. and R. Leland. *An Improved Spectral Load Balancing Algorithm.* Technical Report Tech. Rep. SAND 92-146, Sandia National Laboratories, 1992.

20. Hirsch, C. *Numerical Computation of Internal and External Flows*, *2*, chapter 17, 309–339. Wiley and Sons, 1992.

21. Hirsch, C. *Numerical Computation of Internal and External Flows*, *1*, chapter 4,5,6, 167–266. Wiley and Sons, 1992.

22. Hwang, K. *Advanced Computer Architecture*. McGraw-Hill, 1993.

23. Kumar, V., et al. *Introduction to Parallel Computing*. The Benjamin Cummings Publishing Company, 1994.

24. Löhner, R. and J. Camberosand M. Merriam. "Parallel Unstructured Grid Generation." *Unstructured Scientific Computation on Scalable Multiprocessors*, edited by P. Mehrotra, et al. MIT Press, 1992.

25. Mavriplis, D., et al. "Implementation of a Parallel Unstructured Euler Solver on Shared and Distributed Memory Architectures," *IEEE*, 132–139 (1992).

26. Miel, G. "Supercomputers and CFD," *Aerospace America*, 32–51 (January 1992).

27. Naik, V., et al. "Implicit CFD Applications on Message Passing Multiprocessor Systems." *Parallel Computational Fluid Dynamics Implementation and Results*. MIT Press, 1992.

28. Scherr, S., "Implementation of an Explicit Navier-Stokes Algorithm on a Distributed Memory Parallel Computer." Presented at the 31st Aerospace Sciences Meeting and Exhibit,Jan 11-14,1993,Reno,NV, 1993.

29. Simon, H. "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Engineering*, *2*(2/3):135–148 (1991).

30. Smith, F. "An Unstructured 2-Dimensional Grid Generator." AERO899 Special Study, Air Force Institute of Technology, 1993.

31. Strang, G. *Linear Algebra and Its Applications* (Third Edition). Harcourt Brace Jovanovich, 1988.

32. Venkatakrishnan, V., "Parallel Implicit Unstructured Grid Euler Solvers." Presented at the 32nd Aerospace Sciences Meeting and Exhibit, Jan 10-13,1994,Reno,NV, 1994.

33. Venkatakrishnan, V. and H. Simon. "A MIMD Implementation of a Parallel Euler Solver for Unstructured Grids," *The Journal of Supercomputing*, *6*:117–137 (1992).

34. Walshaw, C. and M. Berzins. *Parallel Computational Fluid Dynamics '92*. Elsevier, 992.

35. Willebeek-Lemair, M. and A.P. Reeves. "Strategies for Dynamic Load Balancing on Highly Parallel Computers," *IEEE Transactions on Parallel and Distributed Systems*, *4*(9) (September 1993).

36. Zhang, S. and H.C. Huang. "Domain decomposition with overlapping and preconditioned conjugate gradient method," *to appear in Jour. Comp. Math* (1994).

Vita

Captain Jay Graham was born on September 8, 1966, in San Antonio, Texas. He graduated from MacArthur High School in San Antonio in 1985. He attended the U.S. Navel Academy, graduating with a Bachelor of Science in Computer Science on May 31,1989. Upon graduate, he received an inter-service transfer to the U.S. Air Force and was commissioned as a Second Lieutenant.

Capt Graham's first assignment was at Arnold Engineering Development Center located near Tullahoma, Tennessee. He was responsible for the management and implementation of a multi-million dollar scientific computer enhancement project designed to keep the Air Force's primary aerodynamic test facilities on the computational leading edge of technology into the next century.

In May 1993, he was reassigned as a student in the Graduate School of Engineering at the Air Force Institute of Technology (AFIT) located at Wright-Patterson AFB, Ohio.

Permanent address: 3565 Knollwood Drive
Beavercreek, OH 45432

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Master's Thesis

4. TITLE AND SUBTITLE
Parallelization of the 2-D Roe Scheme on the Intel Paragon

5. FUNDING NUMBERS

6. AUTHOR(S)
John R. Graham III

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/94D-04

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This study presented a methodology for determining the general performance characteristics of a computational fluid dynamics (CFD) algorithm on the Intel Paragon. By performing a rigorous time complexity analysis of a parallel CFD algorithm, the general performance could be characterized before the code was actually parallelized. This was shown by implementing a serial version of the 2-D Roe Scheme on the Paragon. This explicit code was parallelized by the addition of generic yet efficient routines that decomposed the domain, automatically adjusted partition indices, and performed 2-D and 3-D buffer exchanges. Additionally, efficient global routines available for the Paragon were used in order to reduce the overall complexity of the parallel implementation. Comparison of the predicted performance and the measured performance showed that for the Roe Scheme, the general performance characteristics on the Intel Paragon could be accurately predicted. While a complexity analysis of the Beam-Warming algorithm was performed, a working parallel implementation on the Paragon was not completed.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel CFD, Roe, Reimann Solver, Intel Paragon	99
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL